[Date Prev][Date Next][Thread Prev][Thread Next][Date Index][Thread Index]

An improved version of SRFI 54

- *To*: srfi-54@xxxxxxxxxxxxxxxx
- *Subject*: An improved version of SRFI 54
- *From*: ChurlSoo Joo <initerm@xxxxxxxxx>
- *Date*: Wed, 16 Sep 2009 08:42:33 +0200
- *Delivered-to*: srfi-54@xxxxxxxxxxxxxxxx
- *User-agent*: Gnus/5.110011 (No Gnus v0.11) XEmacs/21.5-b29 (darwin)

I've improved this SRFI in a private capacity.
This is for this SRFI's users and for the record.

--
Joo

```
Title

CAT


Author

Joo ChurlSoo


Changes

The optional arguments of the CAT procedure were divided into three groups;
arguments only for the number type of <object>, arguments for all types except
the number type of <object>, and arguments for all types of <object>.  This
complexity can make users confused.
Those of this revised SRFI are divided into two groups; arguments only for the
number type of <object> and arguments for all types of <object>.  This
simplicity also makes <writer> to be able to substitute for <converter>.
The <precision> actually serves as ~G of Common Lisp's FORMAT and %G of C's
PRINTF.  The <point> of this revised one, an additional optional argument,
serves as ~F or ~E of Common Lisp's FORMAT and %F or %E of C's PRINTF.
The specifications of <take> and <separator> are changed to support diverse
functions.
The CAT procedure of this revised one is extended in following aspects
compared with that of the old one:
1. All optional arguments can be applied to the number type of <object>.
2. The default value of <writer> is DISPLAY procedure.
3. The <precision> is a non-negative inexact integer.
4. An optional argument, <point> ('fixnum or 'flonum), is added.
5. The <take> is changed from a list to a pair, and its elements are exact
   integers, strings, or procedures.
6. The second element of <separator> is changed from a positive exact integer
   to a non-zero exact integer.
7. Infinities, nans, and -0.0 of R6RS are supported.
8. The <pipe> is integrated into the <take>.
9. The <converter> is removed (incompatible change).


Abstract

This SRFI introduces the CAT procedure that takes one object as the first
argument and accepts a variable number of optional arguments, and returns a
string.


Rationale

It is difficult to gain a complete consensus for the design of a generic
formatting procedure that performs a variety of functions provided in C's
```

PRINTF and Common lisp's FORMAT.
One of such ways would be to devise a free non-sequent method that easily handles optional arguments, in contrast to the conventional fixed sequent method, in order to obtain a handy optional and functional interface.

Specification

```
(CAT <object>
     [<port>] [<writer>] [<width>] [<char>] [<take>] [<separator>]
     [<precision%>] [<point%>] [<radix%>] [<sign%>] [<exactness%>]
     [<string>] ...)
```

The <point%> <precision%> <radix%> <sign%> <exactness%> are effective only for the number type of <object>.
Except <string>s, the order of all other optional arguments does not matter.
The CAT processes the optional arguments in the following order; <writer>, <exactness>, <point>, <precision>, <radix>, <separator>, <sign>, <take>, <width>, <char>, <port>.

1. The <object> is any Scheme object.

2. The <width> is an exact integer whose absolute value specifies the width of the resulting string. When the resulting string has fewer characters than the absolute value of <width>, it is placed rightmost with the rest being padded with <char>s, if <width> is positive, or it is placed leftmost with the rest being padded with <char>s, if <width> is negative. On the other hand, when the resulting string has more characters than the absolute value of <width>, the <width> is ignored. The default value is 0.

3. The <writer> is a procedure of two arguments; <object> and a string port. It writes <object> to the string port. The default value is DISPLAY procedure. If you want any objects to be displayed in your own way, you have to define your own <writer>. Otherwise, they are displayed simply in their evaluated forms. When <writer> except DISPLAY and WRITE procedures is used, the optional arguments that are effective only for the number type of <object> become ineffective.

4. The <port> is an output port or a boolean. If an output port is specified, the resulting string is output into the port. If <port> is #t, the output port is current output port. If <port> is #f, the resulting string is returned. The default value is #f.

5. The <char> is a padding character. The default value is #\space.

6. The <take> is a list whose elements are one or two exact integers or strings or procedures, or a pair whose car and cdr values are exact integers or strings or procedures;
   m and n, and the absolute values of m and n are M and N, respectively.
   First,
   When the first element is an exact integer, the resulting string takes from the left m-characters, if it is positive, or all the characters but M-characters, if non-positive.
   When the first element is a string, the string is prefixed.
   When the first element is a procedure, the procedure takes a string argument and returns a string as a pipe.
   Second,
   1. In case that <take> is a list of single element:
      The resulting string is returned.
   2. In case that <take> is a list of two elements:
      When the second element is an exact integer, the resulting string takes from the right n-characters of the string that is processed by the first element, if it is non-negative, or all the characters but N-characters, if negative.
      When the second element is a string, the string is postfixed to the string that is processed by the first element.
      When the second element is a procedure, the procedure takes the string processed by the first element as an argument and returns a string.
   3. In case that <take> is a pair:
      When the cdr element is an exact integer, the other resulting string takes from the right n-characters of the initial string that is not processed by the first element, if it is non-negative, or all the characters but N-characters, if negative. Then, the two strings are concatenated.

When the cdr element is a string, the other resulting string is made by postfixing the element to the initial string that is not processed by the first element. Then, the two strings are concatenated.
When the cdr element is a procedure, the other resulting string is made by the procedure. The procedure takes the initial string that is not processed by the first element, and returns the other resulting string. Then, the two strings are concatenated.

7. The <separator> is a list whose first element is a character serving as a separator and second element is a non-zero exact integer; n, and the absolute value of n is N. The resulting string is separated in every N-characters of the resulting string from right end, if n is positive, or from left end, if n is negative. Even if n is a negative integer, its absolute value is used for the number type of <object>. When the integer is omitted, the <separator> is effective only for the number type of <object> and its default value is 3.

8. The <point> is a symbol: fixnum or flonum. Each returns a string of decimal fraction or exponential representation.

9. The <precision> is a positive inexact integer that specifies the number of decimal digits after a decimal point.

10. The <radix> is a symbol: binary, octal, decimal, or hexadecimal. Each radix sign except decimal is prefixed to the resulting string. The default value is decimal.

11. If <sign> is a symbol that takes the form of 'sign, and <object> is a positive number without a positive sign, the positive sign is prefixed to the resulting string.

12. The <exactness> is a symbol: exact or inexact. Each returns a string of exact or inexact representation.

13. The <string>s are appended to the resulting string.

Examples

```
(cat 129.995 -10 2.)            -> "130.00    "
(cat 129.995 10 2.)             -> "    130.00"
(cat 129.995 2.)                -> "130.00"
(cat 129 10 #\* 'octal 'sign)           -> "****#o+201"
(cat 129 10 #\0 'octal 'sign)           -> "#o+0000201"
(cat 10.5 'octal)                           -> "#i#o25/2"
(cat 10.5 'octal 'exact)                    -> "#o25/2"
(cat 10.5 'octal `(,string-upcase))         -> "#I#O25/2"
(cat 123000000 'flonum)         -> "1.23e+8"
(cat 1.23456789e+25 'fixnum)    -> "12345678900000000000000000.0"
(cat 129.995 10 2. 'sign '("$"))                -> "  $+130.00"
(cat 129.995 10 2. 'sign '("$" -3))             -> "     $+130"
(cat 129.995 10 2. '("The number is " "."))     -> "The number is 130.00."
(cat "abcdefg" '(3 . 1))        -> "abcg"
(cat "abcdefg" '(3 1))          -> "c"
(cat 123456789 'sign '(#\,))            -> "+123,456,789"
(cat "abcdefg" 'sign '(#\,))            -> "abcdefg"
(cat "abcdefg" 'sign '(#\: 2))          -> "a:bc:de:fg"
(cat "abcdefg" 'sign '(#\: -2))         -> "ab:cd:ef:g"
(cat '(#\a "str" s))                  -> "(a str s)"
(cat '(#\a "str" s) '(-1 -1))         -> "a str s"
(cat '(#\a "str" s) write)            -> "(#\\a \"str\" s)"
(cat 'String (current-output-port) 10)   ->     String
(cat 'String #t 10)                       ->     String

(define-record-type :example
  (make-example num str)
  example?
  (num get-num set-num!)
  (str get-str set-str!))

(define (record-writer object string-port)
  (if (example? object)
      (begin (display (get-num object) string-port)
             (display "-" string-port)
             (display (get-str object) string-port))
      (display object string-port)))
```

```
(define (record-display object string-port)
  (display (get-num object) string-port)
  (display "-" string-port)
  (display (get-str object) string-port))

(define ex (make-example 123 "string"))
(cat ex 20)                     -> "  #<struct::example>"
(cat ex 20 record-writer)       -> "          123-string"
(cat "str" 20 record-writer)    -> "                  str"

(let ((plus 12345678.901)
      (minus -123456.789)
      (ex (make-example 1234 "ex"))
      (file "today.txt"))
  (for-each (lambda (x y)
              (cat x #t 10)
              (cat y #t 15 (if (example? y) record-display display) 2. '(#\,))
              (newline))
            (list "plus: " "minus: " "net: " "ex: " "file: ")
            (list plus minus (+ plus minus) ex file)))
->
    plus:   12,345,678.90
   minus:     -123,456.79
     net:   12,222,222.11
      ex:         1234-ex
    file:       today.txt


Reference Implementation

The implementation below requires SRFI 6 (Basic string ports).
;; For compatibility with R6RS
;; (define inexact->exact exact)
;; (define exact->inexact inexact)

(define (object->string object writer)
  (get-output-string
   (let ((str-port (open-output-string)))
     (writer object str-port)
     str-port)))

(define (take-both-end str take)       ;too slow to use `string-append' twice.
  (let ((left (car take)))
    (if (null? (cdr take))
        (cond
         ((string? left) (string-append left str))
         ((procedure? left) (left str))
         ((zero? left) str)
         (else (let ((len (string-length str)))
                 (if (positive? left)
                     (if (< left len)
                         (substring str 0 left)
                         str)
                     (if (positive? (+ len left))
                         (substring str (abs left) len)
                         "")))))
        (if (list? take)
            (let ((right (cadr take)))
              (cond
               ((string? right)
                (cond
                 ((string? left) (string-append left str right))
                 ((procedure? left) (string-append (left str) right))
                 ((zero? left) (string-append str right))
                 (else (string-append (let ((len (string-length str)))
                                        (if (positive? left)
                                            (if (< left len)
                                                (substring str 0 left)
                                                str)
                                            (if (positive? (+ len left))
                                                (substring str (abs left) len)
                                                "")))
                                      right))))
               ((procedure? right)
```

```
                (cond
                 ((string? left) (right (string-append left str)))
                 ((procedure? left) (right (left str)))
                 ((zero? left) (right str))
                 (else (right (let ((len (string-length str)))
                                (if (positive? left)
                                    (if (< left len)
                                        (substring str 0 left)
                                        str)
                                    (if (positive? (+ len left))
                                        (substring str (abs left) len)
                                        ""))))))))
               ((zero? right) "")
               (else (let* ((lt-str
                             (cond
                              ((string? left) (string-append left str))
                              ((procedure? left) (left str))
                              ((zero? left) str)
                              (else (let ((len (string-length str)))
                                      (if (positive? left)
                                          (if (< left len)
                                              (substring str 0 left)
                                              str)
                                          (if (positive? (+ len left))
                                              (substring str (abs left) len)
                                              "")))))))
                            (lt-len (string-length lt-str)))
                      (if (negative? right)
                          (if (positive? (+ lt-len right))
                              (substring lt-str 0 (+ lt-len right))
                              "")
                          (if (< right lt-len)
                              (substring lt-str (- lt-len right) lt-len)
                              lt-str))))))
          (let ((right (cdr take)))
            (cond
             ((string? right)
              (cond
               ((string? left) (string-append left str str right))
               ((procedure? left) (string-append (left str) str right))
               ((zero? left) (string-append str str right))
               (else (string-append (let ((len (string-length str)))
                                      (if (positive? left)
                                          (if (< left len)
                                              (substring str 0 left)
                                              str)
                                          (if (positive? (+ len left))
                                              (substring str (abs left) len)
                                              "")))
                                    str right))))
             ((procedure? right)
              (cond
               ((string? left) (string-append left str (right str)))
               ((procedure? left) (string-append (left str) (right str)))
               ((zero? left) (string-append str (right str)))
               (else (string-append (let ((len (string-length str)))
                                      (if (positive? left)
                                          (if (< left len)
                                              (substring str 0 left)
                                              str)
                                          (if (positive? (+ len left))
                                              (substring str (abs left) len)
                                              "")))
                                    (right str)))))
             ((zero? right)
              (cond
               ((string? left) (string-append left str))
               ((procedure? left) (left str))
               ((zero? left) str)
               (else (let ((len (string-length str)))
                       (if (positive? left)
                           (if (< left len)
                               (substring str 0 left)
                               str)
                           (if (positive? (+ len left))
```

```
                                (substring str (abs left) len)
                                ""))))))
                    (else
                     (let ((len (string-length str)))
                       (string-append (cond
                                       ((string? left) (string-append left str))
                                       ((procedure? left) (left str))
                                       ((zero? left) str)
                                       (else (if (positive? left)
                                                 (if (< left len)
                                                     (substring str 0 left)
                                                     str)
                                                 (if (positive? (+ len left))
                                                     (substring str (abs left) len)
                                                     ""))))
                                      (if (negative? right)
                                          (if (positive? (+ len right))
                                              (substring str 0 (+ len right))
                                              "")
                                          (if (< right len)
                                              (substring str (- len right) len)
                                              str)))))))))

(define (str-char-index str char start end)
  (let lp ((n start))
    (if (= n end)
        #f
        (if (char=? char (string-ref str n))
            n
            (lp (+ n 1))))))

(define (str-numeric-index str start end)
  (let lp ((n start))
    (if (= n end)
        #f
        (if (char-numeric? (string-ref str n))
            n
            (lp (+ n 1))))))

(define (str-numeric? str start end)
  (let lp ((n start))
    (if (= n end)
        #t
        (if (char-numeric? (string-ref str n))
            (lp (+ n 1))
            #f))))

(define (fixnum-string-separate str sep num sig)
  (let* ((len (string-length str))
         (dot-index (str-char-index str #\. 1 len)))
    (if dot-index
        (if sig
            (if (and (str-numeric? str 1 dot-index)
                     (str-numeric? str (+ 1 dot-index) len))
                (string-append
                 (apply string-append
                        (let loop ((ini 0)
                                   (pos (+ 1 (let ((pos (remainder
                                                         (- dot-index 1) num)))
                                               (if (zero? pos) num pos)))))
                          (if (< pos dot-index)
                              (cons (substring str ini pos)
                                    (cons sep (loop pos (+ pos num))))
                              (list (substring str ini dot-index)))))
                 "."
                 (apply string-append
                        (let loop ((ini (+ 1 dot-index))
                                   (pos (+ 1 dot-index num)))
                          (if (< pos len)
                              (cons (substring str ini pos)
                                    (cons sep (loop pos (+ pos num))))
                              (list (substring str ini len))))))
                str)
            (if (and (str-numeric? str 0 dot-index)
                     (str-numeric? str (+ 1 dot-index) len))
```

```
                (string-append
                 (apply string-append
                        (let loop ((ini 0)
                                   (pos (let ((pos (remainder dot-index num)))
                                          (if (zero? pos) num pos))))
                          (if (< pos dot-index)
                              (cons (substring str ini pos)
                                    (cons sep (loop pos (+ pos num))))
                              (list (substring str ini dot-index)))))
                 "."
                 (apply string-append
                        (let loop ((ini (+ 1 dot-index))
                                   (pos (+ 1 dot-index num)))
                          (if (< pos len)
                              (cons (substring str ini pos)
                                    (cons sep (loop pos (+ pos num))))
                              (list (substring str ini len))))))
                str))
          (if sig
              (if (str-numeric? str 1 len)
                  (apply string-append
                         (let loop ((ini 0)
                                    (pos (+ 1 (let ((pos (remainder (- len 1)
                                                                    num)))
                                                (if (zero? pos) num pos)))))
                           (if (< pos len)
                               (cons (substring str ini pos)
                                     (cons sep (loop pos (+ pos num))))
                               (list (substring str ini len)))))
                  str)
              (if (str-numeric? str 0 len)
                  (apply string-append
                         (let loop ((ini 0)
                                    (pos (let ((pos (remainder len num)))
                                           (if (zero? pos) num pos))))
                           (if (< pos len)
                               (cons (substring str ini pos)
                                     (cons sep (loop pos (+ pos num))))
                               (list (substring str ini len)))))
                  str)))))

(define (separate str sep num)
  (let ((len (string-length str))
        (n (abs num)))
    (apply string-append
           (let loop ((ini 0)
                      (pos (if (negative? num)
                               n
                               (let ((pos (remainder len n)))
                                 (if (zero? pos) n pos)))))
             (if (< pos len)
                 (cons (substring str ini pos)
                       (cons sep (loop pos (+ pos n))))
                 (list (substring str ini len)))))))

(define (mold str pre)
  (let* ((len (string-length str))
         (ind (str-char-index str #\. 0 len)))
    (if ind
        (let ((d-len (- len (+ ind 1))))
          (cond
           ((= d-len pre) str)
           ((< d-len pre) (string-append str (make-string (- pre d-len) #\0)))
           ;;((char<? #\4 (string-ref str (+ 1 ind pre)))
           ;;(let ((com (expt 10 pre)))
           ;;  (number->string (/ (round (* (string->number str) com)) com))))
           ((or (char<? #\5 (string-ref str (+ 1 ind pre)))
                (and (char=? #\5 (string-ref str (+ 1 ind pre)))
                     (or (< (+ 1 pre) d-len)
                         (memv (string-ref str (+ ind (if (= 0 pre) -1 pre)))
                               '(#\1 #\3 #\5 #\7 #\9)))))
            (apply
             string
             (let* ((minus (char=? #\- (string-ref str 0)))
                    (str (substring str (if minus 1 0) (+ 1 ind pre)))
```

```
                    (char-list
                     (reverse
                      ;;(let lp ((index (- (string-length str) 1))
                      (let lp ((index (- (+ ind pre) (if minus 1 0)))
                               (raise #t))
                        (if (= -1 index)
                            (if raise '(#\1) '())
                            (let ((chr (string-ref str index)))
                              (if (char=? #\. chr)
                                  (cons chr (lp (- index 1) raise))
                                  (if raise
                                      (if (char=? #\9 chr)
                                          (cons #\0 (lp (- index 1) raise))
                                          (cons (integer->char
                                                 (+ 1 (char->integer chr)))
                                                (lp (- index 1) #f)))
                                      (cons chr (lp (- index 1) raise))))))))))
                (if minus (cons #\- char-list) char-list))))
            (else
             (substring str 0 (+ 1 ind pre)))))
          (string-append str "." (make-string pre #\0)))))

  (define (mold-non-finites str pre)
    (let* ((len (string-length str))
           (ind (str-char-index str #\. 1 len))
           (d-len (- len (+ ind 1))))
      (if (char-numeric? (string-ref str (- ind 1)))
          (cond
           ((= d-len pre) str)
           ((< d-len pre) (string-append str (make-string (- pre d-len) #\0)))
           ;;((char<? #\4 (string-ref str (+ 1 ind pre)))
           ;;(let ((com (expt 10 pre)))
           ;;  (number->string (/ (round (* (string->number str) com)) com))))
           ((or (char<? #\5 (string-ref str (+ 1 ind pre)))
                (and (char=? #\5 (string-ref str (+ 1 ind pre)))
                     (or (< (+ 1 pre) d-len)
                         (memv (string-ref str (+ ind (if (= 0 pre) -1 pre)))
                               '(#\1 #\3 #\5 #\7 #\9)))))
            (apply
             string
             (let* ((minus (char=? #\- (string-ref str 0)))
                    (str (substring str (if minus 1 0) (+ 1 ind pre)))
                    (char-list
                     (reverse
                      ;;(let lp ((index (- (string-length str) 1))
                      (let lp ((index (- (+ ind pre) (if minus 1 0)))
                               (raise #t))
                        (if (= -1 index)
                            (if raise '(#\1) '())
                            (let ((chr (string-ref str index)))
                              (if (char=? #\. chr)
                                  (cons chr (lp (- index 1) raise))
                                  (if raise
                                      (if (char=? #\9 chr)
                                          (cons #\0 (lp (- index 1) raise))
                                          (cons (integer->char
                                                 (+ 1 (char->integer chr)))
                                                (lp (- index 1) #f)))
                                      (cons chr (lp (- index 1) raise))))))))))
                (if minus (cons #\- char-list) char-list))))
            (else
             (substring str 0 (+ 1 ind pre))))
           (error 'cat "infinities or nans cannot have precisions" str pre))))

  (define (e-mold str pre)
    (let* ((len (string-length str))
           (e-index (str-char-index str #\e 1 (- len 1))))
      (if e-index
          (string-append (mold (substring str 0 e-index) pre)
                         (substring str e-index len))
          (mold-non-finites str pre))))

  (define (flonum-mold str pre)
    (let* ((len (string-length str))
           (e-index (str-char-index str #\e 1 (- len 1))))
```

```
     (string-append (mold (substring str 0 e-index) pre)
                    (substring str e-index len))))

 ;; (define (remove-zero str len negative)
 ;;   (if negative
 ;;       (let lp ((n 1))
 ;;      (let ((c (string-ref str n)))
 ;;        (cond
 ;;         ((char=? #\0 c) (lp (+ 1 n)))
 ;;         ((char=? #\. c)
 ;;          (if (= n 2)
 ;;              str
 ;;              (string-append "-" (substring str (- n 1) len))))
 ;;         (else
 ;;          (if (= n 1)
 ;;              str
 ;;              (string-append "-" (substring str n len)))))))
 ;;       (let lp ((n 0))
 ;;      (let ((c (string-ref str n)))
 ;;        (cond
 ;;         ((char=? #\0 c) (lp (+ 1 n)))
 ;;         ((char=? #\. c)
 ;;          (if (= n 1)
 ;;              str
 ;;              (substring str (- n 1) len)))
 ;;         (else
 ;;          (if (zero? n)
 ;;              str
 ;;              (substring str n len)))))))

 (define (real->fixnum-string n)
   (let* ((str (number->string (exact->inexact n)))
          (len (string-length str))
          (e-index (str-char-index str #\e 1 (- len 1))))
     (if e-index
         (let ((e-number (string->number (substring str (+ 1 e-index) len)))
               (d-index (str-char-index str #\. 1 e-index)))
           (if (negative? e-number)
               (if d-index
                   (if (negative? n)
                       (let ((p-number (- (abs e-number) (- d-index 1))))
                         (if (negative? p-number)
                             (let ((pnumber (+ 1 (abs p-number))))
                               (string-append (substring str 0 pnumber)
                                              "."
                                              (substring str pnumber d-index)
                                              (substring str (+ 1 d-index)
                                                         e-index)))
                             (string-append "-0."
                                            (make-string p-number #\0)
                                            (substring str 1 d-index)
                                            (substring str (+ 1 d-index)
                                                       e-index))))
                       (let ((p-number (- (abs e-number) d-index)))
                         (if (negative? p-number)
                             (let ((pnumber (abs p-number)))
                               (string-append (substring str 0 pnumber)
                                              "."
                                              (substring str pnumber d-index)
                                              (substring str (+ 1 d-index)
                                                         e-index)))
                             (string-append "0."
                                            (make-string p-number #\0)
                                            (substring str 0 d-index)
                                            (substring str (+ 1 d-index)
                                                       e-index)))))
                   (if (negative? n)
                       (let ((p-number (- (abs e-number) (- e-index 1))))
                         (if (negative? p-number)
                             (let ((pnumber (+ 1 (abs p-number))))
                               (string-append (substring str 0 pnumber)
                                              "."
                                              (substring str pnumber e-index)))
                             (string-append "-0."
                                            (make-string p-number #\0)
```

```
                                        (substring str 1 e-index))))
                      (let ((p-number (- (abs e-number) e-index)))
                        (if (negative? p-number)
                            (let ((pnumber (abs p-number)))
                              (string-append (substring str 0 pnumber)
                                             "."
                                             (substring str pnumber e-index)))
                            (string-append "0."
                                           (make-string p-number #\0)
                                           (substring str 0 e-index))))))
                (if d-index
                    (let ((p-number (- e-number (- e-index (+ d-index 1)))))
                      (if (negative? p-number)
                          ;; A procedure REMOVE-ZERO is unnecessary
                          ;; due to number->string.
                          ;; 0.00123 -> 00.0123 or 000123
                          ;; -0.00123 -> -00.0123 or -000123
                          ;;(remove-zero (string-append
                          ;;             (substring str 0 d-index)
                          ;;             (substring str (+ 1 d-index)
                          ;;                        (+ 1 d-index e-number))
                          ;;             "."
                          ;;             (substring str (+ 1 d-index e-number)
                          ;;                        e-index))
                          ;;           e-index
                          ;;           (< n 0))
                          (string-append (substring str 0 d-index)
                                         (substring str (+ 1 d-index)
                                                    (+ 1 d-index e-number))
                                         "."
                                         (substring str (+ 1 d-index e-number)
                                                    e-index))
                          ;; A procedure REMOVE-ZERO is unnecessary
                          ;; due to number->string.
                          ;; 0.00123 -> 00.0123 or 000123
                          ;; -0.00123 -> -00.0123 or -000123
                          ;;(remove-zero (string-append
                          ;;             (substring str 0 d-index)
                          ;;             (substring str (+ 1 d-index) e-index)
                          ;;             (make-string p-number #\0)
                          ;;             ".0")
                          ;;           (+ e-index p-number 1)
                          ;;           (< n 0))))
                          (string-append (substring str 0 d-index)
                                         (substring str (+ 1 d-index) e-index)
                                         (make-string p-number #\0) ".0")))
                    (string-append (substring str 0 e-index)
                                   (make-string e-number #\0)
                                   ".0"))))
            (let ((d-index (str-char-index str #\. 1 len)))
              (if (char-numeric? (string-ref str (- d-index 1)))
                  str
                  (error 'cat "infinities or nans cannot be changed into fixed-point numbers" n))))))

(define (non-0-index str start)
  (let lp ((n start))
    (if (char=? #\0 (string-ref str n))
        (lp (+ 1 n))
        n)))

(define (non-0-index-right str end)
  (let lp ((n (- end 1)))
    (if (char=? #\0 (string-ref str n))
        (lp (- n 1))
        n)))

;; (define (non-0-dot-index-right str end)
;;   (let lp ((n (- end 1)))
;;     (let ((c (string-ref str n)))
;;       (if (or (char=? #\0 c) (char=? #\. c))
;;         (lp (- n 1))
;;         n))))

(define (real->flonum-string n)
  (let* ((str (number->string (exact->inexact n)))
```

```
          (len (string-length str))
          (e-index (str-char-index str #\e 1 (- len 1))))
     (if e-index
         str
         (let ((d-index (str-char-index str #\. 1 len)))
           (if (< -1 n 1)
               (if (zero? n)
                   (string-append str "e+0") ;for -0.0 or +0.0
                   (let ((n-index (non-0-index str (+ 1 d-index))))
                     (string-append (if (negative? n) "-" "")
                                    (substring str n-index (+ 1 n-index))
                                    "."
                                    (if (= n-index (- len 1))
                                        "0"
                                        (substring str (+ 1 n-index) len))
                                    "e-"
                                    (number->string (- n-index d-index)))))
               ;;(let ((n-index (non-0-dot-index-right str len)))
               ;;  (if (< n-index d-index)
               (let ((n-index (non-0-index-right str len)))
                 (if (= n-index d-index)
                     (let ((n-index (non-0-index-right str d-index)))
                       (if (char-numeric? (string-ref str n-index))
                           (if (negative? n)
                               (string-append (substring str 0 2)
                                              "."
                                              (if (= n-index 1)
                                                  "0"
                                                  (substring str 2
                                                             (+ 1 n-index)))
                                              "e+"
                                              (number->string (- d-index 2)))
                               (string-append (substring str 0 1)
                                              "."
                                              (if (= n-index 0)
                                                  "0"
                                                  (substring str 1
                                                             (+ 1 n-index)))
                                              "e+"
                                              (number->string (- d-index 1))))
                           (error 'cat "infinities or nans cannot be changed into floating-point numbers" n)))
                     (if (negative? n)
                         (string-append (substring str 0 2)
                                        "."
                                        (substring str 2 d-index)
                                        (substring str (+ 1 d-index)
                                                   (+ 1 n-index))
                                        "e+"
                                        (number->string (- d-index 2)))
                         (string-append (substring str 0 1)
                                        "."
                                        (substring str 1 d-index)
                                        (substring str (+ 1 d-index)
                                                   (+ 1 n-index))
                                        "e+"
                                        (number->string (- d-index 1)))))))))))

;;; define-macro --- first form
;;; (define-macro (let-cat* z vars . body)
;;;   (let ((var (car vars)))
;;;     (let ((n (car var)) (d (cadr var)) (t (caddr var)))
;;;       (if (null? (cdr vars))
;;;       `(let ((,n (if (null? ,z)
;;;                      ,d
;;;                      (if (null? (cdr ,z))
;;;                          (let ((,n (car ,z)))
;;;                            (if ,t ,n (error 'cat "too many argument" ,z)))
;;;                          (error 'cat "too many arguments" ,z)))))
;;;          ,@body)
;;;       (let ((head (gensym)) (tail (gensym)))
;;;         `(let ((,n (if (null? ,z)
;;;                        ,d
;;;                        (let ((,n (car ,z)))
;;;                          (if ,t
;;;                              (begin (set! ,z (cdr ,z)) ,n)
```

```
;;;                               (let lp ((,head (list ,n)) (,tail (cdr ,z)))
;;;                                 (if (null? ,tail)
;;;                                     ,d
;;;                                     (let ((,n (car ,tail)))
;;;                                       (if ,t
;;;                                           (begin
;;;                                             (set! ,z (append (reverse ,head)
;;;                                                              (cdr ,tail)))
;;;                                             ,n)
;;;                                           (lp (cons ,n ,head)
;;;                                               (cdr ,tail)))))))))))
;;;            (let-cat* ,z ,(cdr vars) ,@body)))))))

;;; define-syntax --- second form
(define-syntax let-cat*
  (syntax-rules ()
    ((let-cat* z ((n d t)) bd ...)
     (let ((n (if (null? z)
                  d
                  (if (null? (cdr z))
                      (let ((n (car z)))
                        (if t n (error 'cat "too many argument" z)))
                      (error 'cat "too many arguments" z)))))
       bd ...))
    ((let-cat* z ((n d t) ndt ...) bd ...)
     (let ((n (if (null? z)
                  d
                  (let ((n (car z)))
                    (if t
                        (begin (set! z (cdr z)) n)
                        (let lp ((head (list n)) (tail (cdr z)))
                          (if (null? tail)
                              d
                              (let ((n (car tail)))
                                (if t
                                    (begin (set! z (append (reverse head)
                                                           (cdr tail)))
                                           n)
                                    (lp (cons n head) (cdr tail))))))))))))
       (let-cat* z (ndt ...) bd ...)))))

(define (integer/string/procedure? is)
  (or (integer? is) (string? is) (procedure? is)))

;; (define (part pred ls)
;;   (let lp ((ls ls) (rest '()))
;;     (if (null? ls)
;;      (cons ls (reverse rest))
;;      (if (pred (car ls))
;;          (cons ls (reverse rest))
;;          (lp (cdr ls) (cons (car ls) rest))))))

(define (cat object . rest)
  (if (null? rest)
      (cond
       ((number? object) (number->string object))
       ((string? object) object)
       ((symbol? object) (symbol->string object))
       ((char? object) (string object))
       ((boolean? object) (if object "#t" "#f"))
       (else (object->string object display)))
      ;;(let* ((str-rest (part string? rest))
      ;;       (strs (car str-rest))
      ;;       (rest (cdr str-rest)))
      (let ((strs (let ((n (car rest)))
                    (if (string? n)
                        (let ((n rest)) (set! rest '()) n)
                        (let lp ((head (list n)) (tail (cdr rest)))
                          (if (null? tail)
                              #f
                              (let ((n (car tail)))
                                (if (string? n)
                                    (begin (set! rest (reverse head)) tail)
                                    (lp (cons n head) (cdr tail))))))))))
        (if (null? rest)
```

```
             (apply string-append
                    (cond
                     ((number? object) (number->string object))
                     ((string? object) object)
                     ((symbol? object) (symbol->string object))
                     ((char? object) (string object))
                     ((boolean? object) (if object "#t" "#f"))
                     (else (object->string object display)))
                    strs)
             ;; For macro
             (let-cat* rest
              ((port #f (or (boolean? port) (output-port? port)))
               (writer display (procedure? writer))
               (width 0 (and (integer? width) (exact? width)))
               (char #\space (char? char))
               (precision #f (and (integer? precision)
                                  (inexact? precision)
                                  (not (negative? precision))))
               (point #f (memq point '(fixnum flonum)))
               (radix 'decimal (memq radix '(decimal octal binary hexadecimal)))
               (take #f
                     (and (pair? take)
                          (integer/string/procedure? (car take))
                          (or (null? (cdr take))
                              (if (list? take)
                                  (and (null? (cddr take))
                                       (integer/string/procedure? (cadr take)))
                                  (integer/string/procedure? (cdr take))))))
               (sign #f (eq? 'sign sign))
               (exactness #f (memq exactness '(exact inexact)))
               (separator #f (and (pair? separator)
                                  (char? (car separator))
                                  (or (null? (cdr separator))
                                      (and (list? separator)
                                           (null? (cddr separator))
                                           (integer? (cadr separator))))))))
;;; function --- third form
;;;          (let* ((find-arg
;;;                  (lambda (default pred)
;;;                    (if (null? rest)
;;;                        default
;;;                        (let ((result (car rest)))
;;;                          (if (pred result)
;;;                              (begin (set! rest (cdr rest)) result)
;;;                              (let loop ((head (list result))
;;;                                         (tail (cdr rest)))
;;;                                (if (null? tail)
;;;                                    default
;;;                                    (let ((result (car tail)))
;;;                                      (if (pred result)
;;;                                          (begin
;;;                                            (set! rest
;;;                                                  (append (reverse head)
;;;                                                          (cdr tail)))
;;;                                            result)
;;;                                          (loop (cons result head)
;;;                                                (cdr tail)))))))))))
;;;                  (find-last-arg
;;;                   (lambda (default pred)
;;;                     (if (null? rest)
;;;                         default
;;;                         (if (null? (cdr rest))
;;;                             (if (pred (car rest))
;;;                                 (car rest)
;;;                                 (error 'cat "too many argument" rest))
;;;                             (error 'cat "too many arguments" rest)))))
;;;                  (port
;;;                   (find-arg #f (lambda (x)
;;;                                  (or (boolean? x) (output-port? x)))))
;;;                  (writer (find-arg display procedure?))
;;;                  (width
;;;                   (find-arg 0 (lambda (width)
;;;                                 (and (integer? width) (exact? width)))))
;;;                  (char (find-arg #\space char?))
;;;                  (precision
```

```
;;;                  (find-arg #f (lambda (precision)
;;;                                 (and (integer? precision)
;;;                                      (inexact? precision)
;;;                                      (not (negative? precision))))))
;;;                 (point
;;;                  (find-arg #f (lambda (point)
;;;                                 (memq point '(fixnum flonum)))))
;;;                 (radix
;;;                  (find-arg 'decimal
;;;                            (lambda (radix)
;;;                              (memq radix
;;;                                    '(decimal octal binary hexadecimal)))))
;;;                 (take
;;;                  (find-arg #f
;;;                            (lambda (take)
;;;                              (and (pair? take)
;;;                                   (integer/string/procedure? (car take))
;;;                                   (or (null? (cdr take))
;;;                                       (if (list? take)
;;;                                           (and (null? (cddr take))
;;;                                                (integer/string/procedure?
;;;                                                 (cadr take)))
;;;                                           (integer/string/procedure?
;;;                                            (cdr take))))))))
;;;                 (sign (find-arg #f (lambda (sign) (eq? 'sign sign))))
;;;                 (exactness
;;;                  (find-arg #f (lambda (exactness)
;;;                                 (memq exactness '(exact inexact)))))
;;;                 (separator
;;;                  (find-last-arg
;;;                   #f
;;;                   (lambda (separator)
;;;                     (and (pair? separator)
;;;                          (char? (car separator))
;;;                          (or (null? (cdr separator))
;;;                              (and (list? separator)
;;;                                   (null? (cddr separator))
;;;                                   (integer? (cadr separator))))))))))
;;;
               (let ((str
                 (if (number? object)
                     (if (or (eq? writer display)
                             (eq? writer write))
                         (let* ((inexact-sign
                                 (and (not (eq? radix 'decimal))
                                      (or (and (or precision point)
                                               (error 'cat "non-decimal cannot have a decimal point" radix precision))
                                          (and (inexact? object)
                                               (not (eq? exactness 'exact)))
                                          (eq? exactness 'inexact))
                                      "#i"))
                                (str
                                 (cond
                                  (point
                                   (if (eq? point 'fixnum)
                                       (if precision
                                           (let ((p (inexact->exact precision)))
                                             (if (real? object)
                                                 (mold
                                                  (real->fixnum-string object) p)
                                                 (let ((imag-str
                                                        (real->fixnum-string
                                                         (imag-part object))))
                                                   (string-append
                                                    (mold (real->fixnum-string
                                                           (real-part object)) p)
                                                    ;; for N+0.0i
                                                    (if (char-numeric?
                                                         (string-ref imag-str 0))
                                                        "+" "")
                                                    (mold imag-str p)
                                                    "i"))))
                                           (if (real? object)
                                               (real->fixnum-string object)
                                               (let ((imag-str
                                                      (real->fixnum-string
```

```
                                (imag-part object))))
                           (string-append
                            (real->fixnum-string
                             (real-part object))
                            ;; for N+0.0i
                            (if (char-numeric?
                                 (string-ref imag-str 0))
                                "+" "")
                            imag-str
                            "i"))))
                (if precision ;'flonum
                    (let ((p (inexact->exact precision)))
                      (if (real? object)
                          (flonum-mold
                           (real->flonum-string object) p)
                          (let ((imag-str
                                 (real->flonum-string
                                  (imag-part object))))
                            (string-append
                             (flonum-mold
                              (real->flonum-string
                               (real-part object)) p)
                             ;; for N+0.0i
                             (if (char-numeric?
                                  (string-ref imag-str 0))
                                 "+" "")
                             (flonum-mold imag-str p)
                             "i"))))
                    (if (real? object)
                        (real->flonum-string object)
                        (let ((imag-str
                               (real->flonum-string
                                (imag-part object))))
                          (string-append
                           (real->flonum-string
                            (real-part object))
                           ;; for N+0.0i
                           (if (char-numeric?
                                (string-ref imag-str 0))
                               "+" "")
                           imag-str
                           "i"))))))
          (precision
           (let ((p (inexact->exact precision)))
             (if (real? object)
                 (e-mold (number->string
                          (exact->inexact object)) p)
                 (let ((imag-str
                        (number->string
                         (exact->inexact
                          (imag-part object)))))
                   (string-append
                    (e-mold (number->string
                             (exact->inexact
                              (real-part object))) p)
                    ;; for N+0.0i
                    (if (char-numeric?
                         (string-ref imag-str 0))
                        "+" "")
                    (e-mold imag-str p)
                    "i")))))
          (else
           (number->string
            (cond
             (inexact-sign (inexact->exact object))
             (exactness (if (eq? exactness 'exact)
                            (inexact->exact object)
                            (exact->inexact object)))
             (else object))
            (cdr (assq radix '((decimal . 10)
                               (octal . 8)
                               (hexadecimal . 16)
                               (binary . 2))))))))
        (str (if separator
                 (fixnum-string-separate
```

```
                    str
                    (string (car separator))
                    (if (null? (cdr separator))
                        3 (abs (cadr separator)))
                    (negative? (real-part object)))
                   str))
             (str
              (string-append
               (or inexact-sign "")
               (if (and (eq? exactness 'exact)
                        (or precision point))
                   "#e" "")
               (cdr (assq radix '((decimal . "")
                                  (octal . "#o")
                                  (hexadecimal . "#x")
                                  (binary . "#b"))))
               (if (and sign
                        ;;(positive? (real-part object)))
                        ;; for 0.0
                        (char-numeric? (string-ref str 0)))
                   "+" "")
               str))
             (str (if take (take-both-end str take) str))
             (pad (- (abs width) (string-length str))))
        (cond
         ((<= pad 0) str)
         ((positive? width)
          (if (char-numeric? char)
              (let* ((len (string-length str))
                     (index (str-numeric-index str 0 len)))
                (if index
                    (if (or (zero? index)
                            ;; for infinities and nans
                            (char=? (string-ref str
                                                (- index 1))
                                    #\.))
                        (string-append (make-string pad char)
                                       str)
                        (string-append
                         (substring str 0 index)
                         (make-string pad char)
                         (substring str index len)))
                    (string-append (make-string pad char)
                                   str)))
              (string-append (make-string pad char) str)))
         (else (string-append str (make-string pad char)))))
      (let* ((str (object->string object writer))
             (str (if separator
                      (fixnum-string-separate
                       str
                       (string (car separator))
                       (if (null? (cdr separator))
                           3 (abs (cadr separator)))
                       (negative? (real-part object)))
                      str))
             (str (if take (take-both-end str take) str))
             (pad (- (abs width) (string-length str))))
        (cond
         ((<= pad 0) str)
         ((positive? width)
          (if (char-numeric? char)
              (let* ((len (string-length str))
                     (index (str-numeric-index str 0 len)))
                (if index
                    (if (or (zero? index)
                            ;; for infinities and nans
                            (char=? (string-ref str
                                                (- index 1))
                                    #\.))
                        (string-append (make-string pad char)
                                       str)
                        (string-append
                         (substring str 0 index)
                         (make-string pad char)
                         (substring str index len)))
```

```
                                    (string-append (make-string pad char)
                                                   str)))
                                (string-append (make-string pad char) str)))
                           (else
                            (string-append str (make-string pad char))))))
                  (let* ((str
                          (if (eq? writer display)
                              (cond
                               ((string? object) object)
                               ((symbol? object) (symbol->string object))
                               ((char? object) (string object))
                               ((boolean? object) (if object "#t" "#f"))
                               (else (object->string object writer)))
                              (if (eq? writer write)
                                  (cond
                                   ((symbol? object) (symbol->string object))
                                   ((char? object)
                                    (string-append "#\\" (string object)))
                                   ((boolean? object) (if object "#t" "#f"))
                                   (else (object->string object writer)))
                                  (object->string object writer))))
                         (str (if (and separator
                                       (not (null? (cdr separator))))
                                  (separate str (string (car separator))
                                            (cadr separator))
                                  str))
                         (str (if take (take-both-end str take) str))
                         (pad (- (abs width) (string-length str))))
                    (cond
                     ((<= pad 0) str)
                     ((positive? width) (string-append (make-string pad char)
                                                       str))
                     (else (string-append str (make-string pad char))))))))
            (if port
                (let ((port (if (eq? port #t) (current-output-port) port)))
                  (if strs
                      ;; I don't know which is the best.
                      ;; All of the different implementations have the
                      ;; different performance efficiency of `display' and
                      ;; `string-append'.
                      ;; 1. The simple
                      ;;(display (apply string-append str strs) port)
                      ;; 2. The types of the elements of `strs' are not
                      ;;    checked.
                      ;;(begin
                      ;; (display str port)
                      ;; (for-each (lambda (x) (display x port)) strs))
                      ;; 3. The intermediate
                      ;;(begin
                      ;;  (display str port)
                      ;;  (display (apply string-append strs) port))
                      (display (apply string-append str strs) port)
                      (display str port)))
                (if strs
                    (apply string-append str strs)
                    str)))))))

;;; eof
```

References

[R5RS] Richard Kelsey, William Clinger, and Jonathan Rees: Revised(5) Report on the Algorithmic Language Scheme
http://www.schemers.org/Documents/Standards/R5Rs/

[R6RS] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton von Straaten:
Revised(6) Report on the Algorithmic Language Scheme
http://www.r6rs.org

Copyright

- Prev by Date: **A cat supporting infinities, nans, and -0.0**
- Previous by thread: **A cat supporting infinities, nans, and -0.0**
- Index(es):
  - **Date**
  - **Thread**