Title

Formatting

Author

Joo ChurlSoo

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-54@nospamsrfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2004-03-28
- Draft: 2004-03-22--2004-06-22
- Revised: 2004-04-07
- Revised: 2004-04-12
- Revised: 2004-04-20
- Revised: 2004-04-29
- Revised: 2004-05-27
- Revised: 2004-06-09
- Reference implementation fix: 2006-01-20
- Final: 2004-06-24

Abstract

This SRFI introduces the CAT procedure that converts any object to a string. It takes one object as the first argument and accepts a variable number of optional arguments, unlike the procedure called FORMAT.

Rationale

It is difficult to gain a complete consensus for the design of a generic formatting procedure that performs a variety of necessary functions in addition to essential functions provided in C's PRINTF and Common lisp's FORMAT. One of such ways would be to devise a free (and/or floating) sequence method that easily handles optional arguments, in contrast to the conventional fixed sequence method, in order to obtain a handy optional and functional interface. With the concept of free sequencing, the CAT procedure is then defined, not to process optional arguments with default values, but to process default values with optional arguments.

Issues

In converting a number to a string, it has not been tried to uniformly express the exactness of a number. The CAT procedure makes it possible for the user to prefix an exact sign to the resulting string as well as not to prefix it to the resulting string as conventionally used when an exact number is made to have a decimal point. An inexact sign is prefixed to the resulting string in addition to an radix sign when an inexact number is converted to a non-decimal.

Specification

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

<suffix%>: effective only for the number type of <object>.
<suffix$>: effective for all types except the number type of <object>.
<suffix*>: effective for all types of <object>.
<object> is any Scheme object.
<exactness> is a symbol: exact or inexact.
<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.
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.
<precision> is an inexact integer whose absolute value specifies the number of decimal digits after a decimal point. If <precision> is a non-negative integer, an exact sign is prefixed to the resulting string as needed.
<separator> is a list whose first element is a character serving as a separator and second element is a positive exact integer. If the integer is n, the resulting string is separated in every n-characters of the resulting string. When the integer is omitted, the default value is 3.
<writer> is a procedure of two arguments; <object> and a string port. It writes <object> to the string port. The default value of <writer> is varied according to the type of <object>. When <object> is a self-evaluating constant, it becomes equivalent to DISPLAY procedure, otherwise, it becomes WRITE 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.
<pipe> is a list which is composed of one or more procedures. Each procedure takes at least one string argument and returns a string. One procedure connects with another as a pipe.
<take> is a list whose elements are two exact integers; n and m, and the absolute values of n and m are N and M, respectively. First, the resulting string takes from the left n-characters, if it is non-negative, or all the characters but N-characters, if negative. Second, it takes from the right m-characters, if it is non-negative, or all the characters but M-characters, if negative. Then, it concatenates two set of characters taken. The second element can be omitted. If omitted, the default value is 0.
<converter> is a pair whose car value is a predicate procedure that checks whether <object> satisfies it, and cdr value is a procedure that takes the <object> as an argument and returns a string. When <object> satisfies the predicate procedure, all optional arguments are ineffective except <width>, <char>, <port>, and <string>.
<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.
<char> is a padding character. The default value is #\space.
<port> is an output port or a boolean. If an output port is specified, the resulting string and <string>s are output into that port and simultaneously returned as a string. If <port> is #t, the output port is current output port. If <port> is #f, the output is only returned as a string. The default value is #f.
<string> is a string that is appended to the resulting string.

The order of all optional arguments does not matter. The CAT procedure processes optional arguments in the following order; <exactness>, <radix>, <precision>, <separator>, <sign> for the number type of <object>, or in the following order; <writer>, <pipe>, <take> for all other types.

Examples

```
(cat 129.995 -10 2.) => "130.00    "

(cat 129.995 10 2.) => "    130.00"

(cat 129.985 10 2.) => "    129.98"

(cat 129.985001 10 2.) => "    129.99"

(cat 129.995 2. 'exact) => "#e130.00"
```

```
(cat 129 -2.) => "129.00"

(cat 129 2.) => "#e129.00"

(cat 129 10 2. #\0 'sign) => "#e+0129.00"

(cat 129 10 2. #\* 'sign) => "*#e+129.00"

(cat 1/3) => "1/3"

(cat 1/3 10 2.) => "    #e0.33"

(cat 1/3 10 -2.) => "      0.33"

(cat 129.995 10 '(#\, 2)) => " 1,29.99,5"

(cat 129995 10 '(#\,) 'sign) => "  +129,995"

(cat (cat 129.995 0.) '(0 -1)) => "130"

(cat 99.5 10 'sign 'octal) => "#i#o+307/2"

(cat 99.5 10 'sign 'octal 'exact) => "  #o+307/2"

(cat #x123 'octal 'sign) => "#o+443"

(cat #x123 -10 2. 'sign #\*) => "#e+291.00*"

(cat -1.2345e+15+1.2355e-15i 3.) => "-1.234e15+1.236e-15i"

(cat 1.2345e+15 10 3. 'sign) => " +1.234e15"

(cat "string" -10) => "string    "

(cat "string" 10 (list string-upcase)) => "    STRING"

(cat "string" 10 (list string-upcase) '(-2)) => "      RING"

(cat "string" 10 `(,string-titlecase) '(2 3)) => "     Sting"

(cat "string" `(,string-reverse ,string-upcase) => "GNIRTS"

(cat #\a 10) => "         a"

(cat 'symbol 10) => "    symbol"

(cat '#(#\a "str" s)) => "#(#\\a \"str\" s)"

(cat '(#\a "str" s)) => "(#\\a \"str\" s)"

(cat '(#\a "str" s) #t) => (#\a "str" s)"(#\\a \"str\" s)"

(cat '(#\a "str" s) (current-output-port)) => (#\a "str" s)"(#\\a \"str\" s)"

(cat 3 (cat 's) " " (cat "str" write)) => "3s \"str\""

(cat 3 #t (cat 's) " " (cat "str" write)) => 3s "str""3s \"str\""

(cat 3 #t (cat 's #t) " " (cat "str" write)) => s3s "str""3s \"str\""


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

```
            (str get-str set-str!))
        (define ex (make-example 123 "string"))
        (define (record->string object)
          (cat (get-num object) "-" (get-str object)))
        (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))
                ((or (and (or (string? object)
                              (char? object)
                              (boolean? object))
                          display)
                     write) object string-port)))
        ex => '#{:example}
        (cat ex) => "#{:example}"
        (cat ex 20 record-writer) => "          123-string"
        (cat ex 20 record-writer
             `(,(cut string-delete char-set:digit <>)
               ,string-upcase ,string-reverse)
             '(0 -1) #\-) => "--------------GNIRTS"
        (cat "string" 20 record-writer
             (list string-upcase) '(2 3) #\-) => "---------------STING"
        (cat 12 20 record-writer 3.) => "            #e12.000"
        (cat ex 20 (cons example? record->string)) => "          123-string"
        (cat ex 20 (cons example? record->string)
             `(,(cut string-delete char-set:digit <>)
               ,string-upcase ,string-reverse)
             '(0 -1) #\-) => "----------123-string"
        (cat "string" 20 (cons example? record->string)
             (list string-upcase) '(2 3) #\-) => "---------------STING"
        (cat 12 20 (cons example? record->string) -3.) => "              12.000"
```

Implementation

The implementation below requires SRFI 6 (Basic string ports) and SRFI 23 (Error reporting mechanism).

```
(define (cat object . rest)
  (let* ((str-rest (part string? rest))
         (str-list (car str-rest))
         (rest-list (cdr str-rest)))
    (if (null? rest-list)
        (apply string-append
               (cond
                ((number? object) (number->string object))
                ((string? object) object)
                ((char? object) (string object))
                ((boolean? object) (if object "#t" "#f"))
                ((symbol? object) (symbol->string object))
                (else
                 (get-output-string
                  (let ((str-port (open-output-string)))
                    (write object str-port)
                    str-port))))
               str-list)
        (alet-cat* rest-list
          ((width 0 (and (integer? width) (exact? width)))
           (port #f (or (boolean? port) (output-port? port))
                 (if (eq? port #t) (current-output-port) port))
           (char #\space (char? char))
           (converter #f (and (pair? converter)
                     (procedure? (car converter))
                     (procedure? (cdr converter))))
   (precision #f (and (integer? precision)
                     (inexact? precision)))
   (sign #f (eq? 'sign sign))
   (radix 'decimal
          (memq radix '(decimal octal binary hexadecimal)))
   (exactness #f (memq exactness '(exact inexact)))
   (separator #f (and (list? separator)
                     (< 0 (length separator) 3)
                     (char? (car separator))
                     (or (null? (cdr separator))
                         (let ((n (cadr separator)))
                           (and (integer? n) (exact? n)
                                (< 0 n))))))
   (writer #f (procedure? writer))
   (pipe #f (and (list? pipe)
                (not (null? pipe))
                (every? procedure? pipe)))
   (take #f (and (list? take)
                (< 0 (length take) 3)
                (every? (lambda (x)
                          (and (integer? x) (exact? x)))
                        take))))
  (let* ((str
          (cond
           ((and converter
                 ((car converter) object))
            (let* ((str ((cdr converter) object))
                   (pad (- (abs width) (string-length str))))
              (cond
               ((<= pad 0) str)
               ((< 0 width) (string-append (make-string pad char) str))
               (else (string-append str (make-string pad char))))))
           ((number? object)
            (and (not (eq? radix 'decimal)) precision
                 (error "cat: non-decimal cannot have a decimal point"))
            (and precision (< precision 0) (eq? exactness 'exact)
                 (error "cat: exact number cannot have a decimal point without exact sign"))
            (let* ((exact-sign (and precision
                                    (<= 0 precision)
                                    (or (eq? exactness 'exact)
                                        (and (exact? object)
                                             (not (eq? exactness
                                                       'inexact))))
                                    "#e"))
                   (inexact-sign (and (not (eq? radix 'decimal))
                                      (or (and (inexact? object)
                                               (not (eq? exactness
                                                         'exact)))
                                          (eq? exactness 'inexact))
                                      "#i"))
                   (radix-sign (cdr (assq radix
                                          '((decimal . #f)
                                            (octal . "#o")
                                            (binary . "#b")
                                            (hexadecimal . "#x")))))
                   (plus-sign (and sign (< 0 (real-part object)) "+"))
                   (exactness-sign (or exact-sign inexact-sign))
                   (str
                    (if precision
                        (let ((precision (inexact->exact
                                          (abs precision)))
                              (imag (imag-part object)))
                          (if (= 0 imag)
                              (e-mold object precision)
                              (string-append
                               (e-mold (real-part object) precision)
                               (if (< 0 imag) "+" "")
                               (e-mold imag precision)
                               "i")))
                        (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)
                                            (binary . 2)
                                            (hexadecimal . 16)))))))
                   (str
                    (if (and separator
                             (not (or (and (eq? radix 'decimal)
                                           (str-index str #\e))
                                      (str-index str #\i)
                                      (str-index str #\/))))
                        (let ((sep (string (car separator)))
                              (num (if (null? (cdr separator))
                                       3 (cadr separator)))
                              (dot-index (str-index str #\.)))
                          (if dot-index
                              (string-append
                               (separate (substring str 0 dot-index)
                                         sep num (if (< object 0)
                                                     'minus #t))
                               "."
                               (separate (substring
                                          str (+ 1 dot-index)
                                          (string-length str))
                                         sep num #f))
                              (separate str sep num (if (< object 0)
                                                        'minus #t))))
                        str))
                   (pad (- (abs width)
                           (+ (string-length str)
                              (if exactness-sign 2 0)
                              (if radix-sign 2 0)
                              (if plus-sign 1 0))))
                   (pad (if (< 0 pad) pad 0)))
              (if (< 0 width)
                  (if (char-numeric? char)
                      (if (< (real-part object) 0)
                          (string-append (or exactness-sign "")
                                         (or radix-sign "")
                                         "-"
                                         (make-string pad char)
                                         (substring str 1
                                                    (string-length
                                                     str)))
                          (string-append (or exactness-sign "")
                                         (or radix-sign "")
                                         (or plus-sign "")
                                         (make-string pad char)
                                         str))
                      (string-append (make-string pad char)
                                     (or exactness-sign "")
                                     (or radix-sign "")
```

```
                                 (or plus-sign "")
                                 str))
              (string-append (or exactness-sign "")
                             (or radix-sign "")
                             (or plus-sign "")
                             str
                             (make-string pad char)))))
       (else
        (let* ((str (cond
                     (writer (get-output-string
                              (let ((str-port
                                     (open-output-string)))
                                (writer object str-port)
                                str-port)))
                     ((string? object) object)
                     ((char? object) (string object))
                     ((boolean? object) (if object "#t" "#f"))
                     ((symbol? object) (symbol->string object))
                     (else (get-output-string
                            (let ((str-port (open-output-string)))
                              (write object str-port)
                              str-port)))))
               (str (if pipe
                        (let loop ((str ((car pipe) str))
                                   (fns (cdr pipe)))
                          (if (null? fns)
                              str
                              (loop ((car fns) str)
                                    (cdr fns))))
                        str))
               (str
                (if take
                    (let ((left (car take))
                          (right (if (null? (cdr take))
                                     0 (cadr take)))
                          (len (string-length str)))
                      (define (substr str beg end)
                        (let ((end (cond
                                    ((< end 0) 0)
                                    ((< len end) len)
                                    (else end)))
                              (beg (cond
                                    ((< beg 0) 0)
                                    ((< len beg) len)
                                    (else beg))))
                          (if (and (= beg 0) (= end len))
                              str
                              (substring str beg end))))
                      (string-append
                       (if (< left 0)
                           (substr str (abs left) len)
                           (substr str 0 left))
                       (if (< right 0)
                           (substr str 0 (+ len right))
                           (substr str (- len right) len))))
                    str))
               (pad (- (abs width) (string-length str))))
          (cond
           ((<= pad 0) str)
           ((< 0 width) (string-append (make-string pad char) str))
           (else (string-append str (make-string pad char))))))))
     (str (apply string-append str str-list)))
(and port (display str port))
str)))))
```

```
(define-syntax alet-cat*         ; borrowed from SRFI-86
  (syntax-rules ()
    ((alet-cat* z (a . e) bd ...)
     (let ((y z))
       (%alet-cat* y (a . e) bd ...)))))

(define-syntax %alet-cat*                ; borrowed from SRFI-86
  (syntax-rules ()
    ((%alet-cat* z ((n d t ...)) bd ...)
     (let ((n (if (null? z)
                  d
                  (if (null? (cdr z))
                      (wow-cat-end z n t ...)
                      (error "cat: too many arguments" (cdr z))))))
       bd ...))
    ((%alet-cat* z ((n d t ...) . e) bd ...)
     (let ((n (if (null? z)
                  d
                  (wow-cat! z n d t ...))))
       (%alet-cat* z e bd ...)))
    ((%alet-cat* z e bd ...)
     (let ((e z)) bd ...))))

(define-syntax wow-cat!                  ; borrowed from SRFI-86
  (syntax-rules ()
    ((wow-cat! z n d)
     (let ((n (car z)))
       (set! z (cdr z))
       n))
    ((wow-cat! z n d t)
     (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)))))))))
    ((wow-cat! z n d t ts)
     (let ((n (car z)))
       (if t
           (begin (set! z (cdr z)) ts)
           (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))) ts)
                       (lp (cons n head) (cdr tail)))))))))
    ((wow-cat! z n d t ts fs)
     (let ((n (car z)))
       (if t
           (begin (set! z (cdr z)) ts)
           (begin (set! z (cdr z)) fs))))))

(define-syntax wow-cat-end               ; borrowed from SRFI-86
  (syntax-rules ()
    ((wow-cat-end z n)
     (car z))
    ((wow-cat-end z n t)
     (let ((n (car z)))
       (if t n (error "cat: too many argument" z))))
    ((wow-cat-end z n t ts)
     (let ((n (car z)))
       (if t ts (error "cat: too many argument" z))))
    ((wow-cat-end z n t ts fs)
     (let ((n (car z)))
       (if t ts fs)))))

(define (str-index str char)
  (let ((len (string-length str)))
    (let lp ((n 0))
      (and (< n len)
           (if (char=? char (string-ref str n))
               n
               (lp (+ n 1)))))))

(define (every? pred ls)
  (let lp ((ls ls))
    (or (null? ls)
        (and (pred (car ls))
             (lp (cdr ls))))))

(define (part pred ls)
  (let lp ((ls ls) (true '()) (false '()))
    (cond
     ((null? ls) (cons (reverse true) (reverse false)))
     ((pred (car ls)) (lp (cdr ls) (cons (car ls) true) false))
     (else (lp (cdr ls) true (cons (car ls) false))))))

(define (e-mold num pre)
  (let* ((str (number->string (exact->inexact num)))
         (e-index (str-index str #\e)))
    (if e-index
        (string-append (mold (substring str 0 e-index) pre)
                       (substring str e-index (string-length str)))
        (mold str pre))))

(define (mold str pre)
  (let ((ind (str-index str #\.)))
    (if ind
        (let ((d-len (- (string-length 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))
                               (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 (separate str sep num opt)
(let* ((len (string-length str))
       (pos (if opt
                (let ((pos (remainder (if (eq? opt 'minus) (- len 1) len)
                                      num)))
                  (if (= 0 pos) num pos))
                num)))
  (apply string-append
         (let loop ((ini 0)
                    (pos (if (eq? opt 'minus) (+ pos 1) pos)))
           (if (< pos len)
               (cons (substring str ini pos)
                     (cons sep (loop pos (+ pos num))))
               (list (substring str ini len)))))))

;;; eof
```

Acknowledgment

I owe much to those who incite me to make this SRFI better. And I must thank Michael Sperber for his encouragement and guidance during the draft period of SRFI 51, as the important part of this SRFI is based on SRFI 51. Without him, neither SRFI 51 nor SRFI 54 would have been finilized. Again, I greatly appreciate his kindness and effort. Finally, I deeply apologize to Francisco Solsona for bringing disgrace upon him against my will early in the draft period.

Copyright

Editor: Francisco Solsona
Last modified: Sun Jan 28 13:40:19 MET 2007