

SRFIs: Libraries

Version 9.4.0.3

October 29, 2026

The Scheme Requests for Implementation (a.k.a. *SRFI*) process allows individual members of the Scheme community to propose libraries and extensions to be supported by multiple Scheme implementations.

Racket is distributed with implementations of many SRFIs, most of which can be implemented as libraries. To import the bindings of SRFI *n*, use

```
(require srfi/n)
```

This document lists the SRFIs that are supported by Racket and provides links to the original SRFI specifications (which are also distributed as part of Racket's documentation).

Contents

SRFI 1: List Library	6
SRFI 2: AND-LET*: an AND with local bindings...	7
SRFI 4: Homogeneous numeric vector datatypes	8
SRFI 5: A compatible let form with signatures and rest arguments	9
SRFI 6: Basic String Ports	12
SRFI 7: Feature-based program configuration language	13
SRFI 8: RECEIVE: Binding to multiple values	14
SRFI 9: Defining Record Types	15
SRFI 11: Syntax for receiving multiple values	16
SRFI 13: String Libraries	17
SRFI 14: Character-set Library	18
SRFI 16: Syntax for procedures of variable arity	19
SRFI 17: Generalized set!	20
SRFI 19: Time Data Types and Procedures	21
SRFI 23: Error reporting mechanism	22
SRFI 25: Multi-dimensional Array Primitives	23

SRFI 26: Notation for Specializing Parameters without Currying	24
SRFI 27: Sources of Random Bits	25
SRFI 28: Basic Format Strings	26
SRFI 29: Localization	27
SRFI 30: Nested Multi-line Comments	28
SRFI 31: A special form rec for recursive evaluation	29
SRFI 34: Exception Handling for Programs	30
SRFI 35: Conditions	31
SRFI 38: External Representation for Data With Shared Structure	32
SRFI 39: Parameter objects	33
SRFI 40: A Library of Streams	34
SRFI 41: Streams	35
SRFI 42: Eager Comprehensions	36
SRFI 43: Vector Library	37
SRFI 45: Primitives for Expressing Iterative Lazy Algorithms	38
SRFI 48: Intermediate Format Strings	39
SRFI 54: Formatting	40

SRFI 57: Records	41
SRFI 59: Vicinity	42
SRFI 60: Integers as Bits	43
SRFI 61: A more general cond clause	44
SRFI 62: S-expression comments	45
SRFI 63: Homogeneous and Heterogeneous Arrays	46
SRFI 64: A Scheme API for test suites	47
SRFI 66: Octet Vectors	48
SRFI 67: Compare Procedures	49
SRFI 69: Basic hash tables	50
SRFI 71: Extended LET-syntax for multiple values	51
SRFI 74: Octet-Addressed Binary Blocks	52
SRFI 78: Lightweight testing	53
SRFI 86: MU & NU simulating VALUES & CALL-WITH-VALUES...	54
SRFI 87: => in case clauses	55
SRFI 98: An interface to access environment variables	56
Index	57

SRFI 1: List Library

```
(require srfi/1)           package: srfi-lite-lib
```

Original specification: SRFI 1

This SRFI works with pairs and lists as in **racket**, which are immutable, so it does not export **set-car!** and **set-cdr!**. The other provided bindings that end in **!** are equivalent to the corresponding bindings without **!**. Functions that are documented in the SRFI in bold (but not bold italic) correspond to **racket** functions, while the others are distinct from same-named **racket** functions.

SRFI 2: AND-LET*: an AND with local bindings...

```
(require srfi/2)      package: srfi-lib
```

Original specification: SRFI 2

SRFI 4: Homogeneous numeric vector datatypes

```
(require srfi/4)      package: srfi-lib
```

Original specification: SRFI 4

This SRFI's reader and printer syntax is not supported. The bindings are also available from [scheme/foreign](#).

SRFI 5: A compatible let form with signatures and rest arguments

```
(require srfi/5)      package: srfi-lib
```

Original specification: SRFI 5

For historical reasons, the SRFI 5 specification document has a restrictive license and is not included in the main Racket distribution.

The implementation in `srfi/5` and this documentation are distributed under the same license as Racket: only the original specification document is restrictively licensed.

```
(let ([id init-expr] ...)
  body ...+)
(let ([id init-expr] ...+ rest-binding)
  body ...+)
(let loop-id ([id init-expr] ... maybe-rest-binding)
  body ...+)
(let (loop-id [id init-expr] ... maybe-rest-binding)
  body ...+)

maybe-rest-binding =
    | rest-binding

rest-binding = rest-id rest-init-expr ...
```

Like `let` from `racket/base`, but extended to support additional variants of named `let`.

As with `let` from `racket/base`, SRFI 5's `let` form conceptually expands to the immediate application of a function to the values of the *init-exprs*: the *ids* are bound in the *bodys* (but not in any *init-exprs* or *rest-init-exprs*), and *loop-id*, if present, is bound in the *bodys* to the function itself, allowing it to be used recursively. An *id* or a *rest-id* can shadow *loop-id*, but the *rest-id* (if given) and all *ids* must be distinct.

SRFI 5's `let` adds support for a syntax like `define`'s function shorthand, which allows the bindings to be written in a syntax resembling an application of the function bound to *loop-id*.

Additionally, SRFI 5's `let` adds support for rest arguments. If a *rest-id* is present, the function bound to *loop-id* (or the conceptual anonymous function, if *loop-id* is not used) will accept an unlimited number of additional arguments after its required by-position arguments, and the *rest-id* will be bound in the *bodys* (but not in any *init-exprs* or *rest-init-exprs*) to a list of those additional arguments. The values of the *rest-init-exprs* are supplied as arguments to the initial, implicit application when the `let` form is evaluated, so the initial value bound to *rest-id* is `(list rest-init-expr ...)`.

Unlike the *kw-formals* of `lambda` and `define` or the *formals* of `case-lambda`, the bindings of SRFI 5's `let`, with or without a *rest-binding*, are always a proper (syntactic) list.

A *rest-binding* can be used with both the define-like and the named-let-like variants of let. It is also possible to use *rest-id* without any *loop-id*; however, as specified in the grammar, at least one *id-init-expr* pair is required in that case. (Otherwise, there would be an ambiguity with the define-like variant).

Examples:

```
; define-like bindings
> (define (factorial n)
    (let (fact [n n] [acc 1])
      (if (zero? n)
          acc
          (fact (sub1 n) (* n acc))))))
> (factorial 5)
120
> (factorial 11)
39916800
; rest arguments with named-let--like bindings
> (let reverse-onto ([lst '(a b c)]
                    tail)
    (if (null? lst)
        tail
        (apply reverse-onto (cdr lst) (car lst) tail)))
'(c b a)
> (let reverse-onto ([lst '(a b c)]
                    tail 'x 'y 'z)
    (if (null? lst)
        tail
        (apply reverse-onto (cdr lst) (car lst) tail)))
'(c b a x y z)
> (let no-evens (lst 1 2 3 4 5)
    (cond
      [(null? lst)
       '()]
      [(even? (car lst))
       (apply no-evens (cdr lst))]
      [else
       (cons (car lst) (apply no-evens (cdr lst)))]))
'(1 3 5)
; rest arguments with define-like bindings
> (let (reverse-onto [lst '(a b c)] tail)
    (if (null? lst)
        tail
        (apply reverse-onto (cdr lst) (car lst) tail)))
'(c b a)
> (let (reverse-onto [lst '(a b c)]    tail 'x 'y 'z)
```

```

      (if (null? lst)
          tail
          (apply reverse-onto (cdr lst) (car lst) tail)))
'(c b a x y z)
> (let (loop [continue? 0] args 'a 'a1 'a2)
    (case continue?
      [(0) (cons args (loop 1 'b))]
      [(1) (cons args (loop 2 'c 'd))]
      [else (list args)]))
'((a a1 a2) (b) (c d))
; rest arguments without any loop-id
> (let ([x 1]
        [y 2]
        z 3 4 5 6 7)
    (list* x y z))
'(1 2 3 4 5 6 7)

```

SRFI 6: Basic String Ports

```
(require srfi/6)      package: srfi-lib
```

Original specification: SRFI 6

This SRFI's bindings are also available in [racket/base](#).

SRFI 7: Feature-based program configuration language

```
(require srfi/7)      package: srfi-lib
```

Original specification: SRFI 7

SRFI 8: RECEIVE: Binding to multiple values

```
(require srfi/8)      package: srfi-lite-lib
```

Original specification: SRFI 8

SRFI 9: Defining Record Types

```
(require srfi/9)      package: srfi-lib
```

Original specification: SRFI 9

SRFI 11: Syntax for receiving multiple values

```
(require srfi/11)      package: srfi-lib
```

Original specification: [SRFI 11](#)

This SRFI's bindings are also available in [racket/base](#), but without support for dotted “rest” bindings.

SRFI 13: String Libraries

```
(require srfi/13)      package: srfi-lite-lib
```

Original specification: SRFI 13

SRFI 14: Character-set Library

```
(require srfi/14)      package: srfi-lite-lib
```

Original specification: SRFI 14

SRFI 16: Syntax for procedures of variable arity

```
(require srfi/16)      package: srfi-lib
```

Original specification: SRFI 16

This SRFI's bindings are also available in [racket/base](#).

SRFI 17: Generalized set!

```
(require srfi/17)      package: srfi-lib
```

Original specification: SRFI 17

SRFI 19: Time Data Types and Procedures

```
(require srfi/19)      package: srfi-lite-lib
```

Original specification: SRFI 19

The date structure produced by this SRFI library is identical to the one provided by `racket/base` in most cases (see `date`).

For backwards compatibility, when an invalid date field value is provided to the SRFI constructor, the constructor will produce a *lax date structure*. A lax date structure is *not* compatible with functions from `racket/base` or `racket/date`. SRFI functions such as `string->date` may return a lax date structure depending on the format string. The predicate `lax-date?` recognizes lax date structures.

As an extension, Racket's implementation of `string->date` supports `~?` as a conversion specifier: it parses one- and two-digit years like `~y` and three- and four-digit years like `~Y`.

Examples:

```
> (string->date "4-1-99" "~d-~m-~?")
(date* 0 0 0 4 1 1999 1 3 #f -18000 0 "")
> (string->date "4-1-1999" "~d-~m-~?")
(date* 0 0 0 4 1 1999 1 3 #f -18000 0 "")
```

```
(lax-date? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a lax date structure. Otherwise, returns `#f`.

Examples:

```
> (lax-date? (make-date 0 19 10 10 14 "bogus" "bogus" 0))
#t
> (lax-date? (make-date 0 19 10 10 14 1 2013 0))
#f
> (lax-date? (string->date "10:21:00" "~H:~M:~S"))
#t
```

SRFI 23: Error reporting mechanism

```
(require srfi/23)      package: srfi-lib
```

Original specification: SRFI 23

This SRFI's bindings are also available in [racket/base](#).

SRFI 25: Multi-dimensional Array Primitives

```
(require srfi/25)      package: srfi-lib
```

Original specification: SRFI 25

SRFI 26: Notation for Specializing Parameters without Currying

```
(require srfi/26)      package: srfi-lib
```

Original specification: SRFI 26

SRFI 27: Sources of Random Bits

```
(require srfi/27)      package: srfi-lib
```

Original specification: SRFI 27

SRFI 28: Basic Format Strings

```
(require srfi/28)      package: srfi-lib
```

Original specification: SRFI 28

This SRFI's bindings are also available in [racket/base](#).

SRFI 29: Localization

```
(require srfi/29)      package: srfi-lite-lib
```

Original specification: SRFI 29

SRFI 30: Nested Multi-line Comments

```
(require srfi/30)      package: srfi-lib
```

Original specification: SRFI 30

This SRFI's syntax is part of Racket's default reader.

SRFI 31: A special form `rec` for recursive evaluation

```
(require srfi/31)      package: srfi-lib
```

Original specification: SRFI 31

SRFI 34: Exception Handling for Programs

```
(require srfi/34)      package: srfi-lib
```

Original specification: SRFI 34

An `else` is recognized as either the one from `racket/base` or as an identifier with the symbolic form `'else` and no binding.

SRFI 35: Conditions

```
(require srfi/35)      package: srfi-lib
```

Original specification: SRFI 35

SRFI 38: External Representation for Data With Shared Structure

```
(require srfi/38)      package: srfi-lib
```

Original specification: SRFI 38

This SRFI's syntax is part of Racket's default reader and printer.

SRFI 39: Parameter objects

```
(require srfi/39)      package: srfi-lib
```

Original specification: SRFI 39

This SRFI's bindings are also available in [racket/base](#).

SRFI 40: A Library of Streams

`(require srfi/40)` `package: srfi-lib`

Original specification: SRFI 40

Superseded by [srfi/41](#).

SRFI 41: Streams

```
(require srfi/41)      package: srfi-lib
```

Original specification: SRFI 41

The `stream-cons` operation from [srfi/41](#) is the same as from [racket/stream](#).

SRFI 42: Eager Comprehensions

```
(require srfi/42)      package: srfi-lib
```

Original specification: SRFI 42

Forms that syntactically detect if recognize both if from [racket/base](#) and if from [mzscheme](#).

SRFI 43: Vector Library

```
(require srfi/43)      package: srfi-lib
```

Original specification: SRFI 43

SRFI 45: Primitives for Expressing Iterative Lazy Algorithms

```
(require srfi/45)      package: srfi-lib
```

Original specification: SRFI 45

Additional binding:

```
(promise? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a promise, `#f` otherwise.

SRFI 48: Intermediate Format Strings

```
(require srfi/48)      package: srfi-lib
```

Original specification: SRFI 48

SRFI 54: Formatting

```
(require srfi/54)      package: srfi-lib
```

Original specification: SRFI 54

SRFI 57: Records

```
(require srfi/57)      package: srfi-lib
```

Original specification: SRFI 57

SRFI 59: Vicinity

```
(require srfi/59)      package: srfi-lib
```

Original specification: SRFI 59

SRFI 60: Integers as Bits

```
(require srfi/60)      package: srfi-lib
```

Original specification: SRFI 60

SRFI 61: A more general cond clause

```
(require srfi/61)      package: srfi-lib
```

Original specification: SRFI 61

SRFI 62: S-expression comments

Original specification: SRFI 62

This SRFI's syntax is part of Racket's default reader (no `require` is needed).

SRFI 63: Homogeneous and Heterogeneous Arrays

```
(require srfi/63)      package: srfi-lib
```

Original specification: SRFI 63

SRFI 64: A Scheme API for test suites

```
(require srfi/64)      package: srfi-lib
```

Original specification: SRFI 64

SRFI 66: Octet Vectors

```
(require srfi/66)      package: srfi-lib
```

Original specification: SRFI 66

SRFI 67: Compare Procedures

```
(require srfi/67)      package: srfi-lib
```

Original specification: SRFI 67

SRFI 69: Basic hash tables

```
(require srfi/69)      package: srfi-lib
```

Original specification: SRFI 69

SRFI 71: Extended LET-syntax for multiple values

```
(require srfi/71)      package: srfi-lib
```

Original specification: SRFI 71

SRFI 74: Octet-Addressed Binary Blocks

```
(require srfi/74)      package: srfi-lib
```

Original specification: SRFI 74

SRFI 78: Lightweight testing

```
(require srfi/78)      package: srfi-lib
```

Original specification: SRFI 78

SRFI 86: MU & NU simulating VALUES & CALL-WITH-VALUES...

```
(require srfi/86)      package: srfi-lib
```

Original specification: SRFI 86

SRFI 87: => in case clauses

```
(require srfi/87)      package: srfi-lib
```

Original specification: SRFI 87

SRFI 98: An interface to access environment variables

```
(require srfi/98)      package: srfi-lib
```

Original specification: SRFI 98

Index

->char-set
:
:char-range
:dispatched
:do
:generator-proc
:integers
:let
:list
:parallel
:port
:range
:real-range
:string
:until
:vector
:while
add-duration
add-duration!
alist-cons
alist-copy
alist-delete
alist-delete!
and
and-let*
any
any?-ec
append
append!
append-ec
append-map
append-map!
append-reverse
append-reverse!
array
array-end
array-rank
array-ref
array-set!
array-start
array?
assoc
assq
assv
begin
break
break!
car
car+cdr
case-lambda
cddadr
cddddr
cdr
char-set
char-set->list
char-set->string
char-set-adjoin
char-set-adjoin!
char-set-any
char-set-complement
char-set-complement!
char-set-contains?
char-set-copy
char-set-count
char-set-cursor
char-set-cursor-next
char-set-delete
char-set-delete!
char-set-diff+intersection
char-set-diff+intersection!
char-set-difference
char-set-difference!
char-set-every
char-set-filter
char-set-filter!
char-set-fold
char-set-for-each
char-set-hash
char-set-intersection
char-set-intersection!
char-set-map
char-set-ref

char-set-size	current-time
char-set-unfold	cut
char-set-unfold!	cute
char-set-union	date->julian-day
char-set-union!	date->modified-julian-day
char-set-xor	date->string
char-set-xor!	date->time-monotonic
char-set:ascii	date->time-tai
char-set:blank	date->time-utc
char-set:digit	date-day
char-set:empty	date-hour
char-set:full	date-minute
char-set:graphic	date-month
char-set:hex-digit	date-nanosecond
char-set:iso-control	date-second
char-set:letter	date-week-day
char-set:letter+digit	date-week-number
char-set:lower-case	date-year
char-set:printing	date-year-day
char-set:punctuation	date-zone-offset
char-set:symbol	date?
char-set:title-case	declare-bundle!
char-set:upper-case	default-random-source
char-set:whitespace	define-record-type
char-set<=	define-stream
char-set=	delay
char-set?	delete
check-substring-spec	delete!
circular-list	delete-duplicates
circular-list?	delete-duplicates!
concatenate	do-ec
concatenate!	dotted-list?
cons	drop
cons*	drop-right
copy-time	drop-right!
count	drop-while
current-country	eager
current-date	eighth
current-julian-day	end-of-char-set?
current-language	error
current-locale-details	every
current-modified-julian-day	every?-ec

f32vector	length+
f64vector	let
fifth	let*-values
filter	let-string-start+end
filter!	let-values
filter-map	library-vicinity
find	list
find-tail	list->char-set
first	list->char-set!
first-ec	list->stream
fold	list->string
fold-ec	list->vector
fold-right	list-copy
fold3-ec	list-ec
for-each	list-index
force	list-ref
format	list-tabulate
format	load-bundle!
fourth	localized-template
generator	lset-adjoin
get-environment-variable	lset-diff+intersection
get-environment-variables	lset-diff+intersection!
get-output-string	lset-difference
getter-with-setter	lset-difference!
guard	lset-intersection
home-vicinity	lset-intersection!
if	lset-union
implementation-vicinity	lset-union!
in-vicinity	lset-xor
iota	lset-xor!
julian-day->date	lset=
julian-day->time-monotonic	make-array
julian-day->time-tai	make-date
julian-day->time-utc	make-kmp-restart-vector
kmp-step	make-list
last	make-parameter
last-ec	make-random-source
last-pair	make-string
<i>lax date structure</i>	make-time
lax-date?	make-vector
lazy	make-vicinity
length	map

map!
 map-in-order
 max-ec
 member
 memq
 memv
 min-ec
 modified-julian-day->date
 modified-julian-day->time-
 monotonic
 modified-julian-day->time-tai
 modified-julian-day->time-utc
 nested
 ninth
 not
 not-pair?
 null-list?
 null?
 open-input-string
 open-output-string
 or
 pair-fold
 pair-fold-right
 pair-for-each
 pair?
 parameterize
 partition
 partition!
 pathname->vicinity
 port->stream
 product-ec
 program
 program-vicinity
 promise?
 proper-list?
 raise
 random-integer
 random-real
 random-source-make-integers
 random-source-make-reals
 random-source-pseudo-randomize!
 random-source-randomize!

random-source-state-ref
 random-source-state-ref!
 random-source?
 read-with-shared-structure
 rec
 receive
 reduce
 reduce-right
 remove
 remove!
 reverse
 reverse!
 reverse-list->string
 reverse-list->vector
 reverse-vector->list
 s16vector
 s32vector
 s64vector
 s8vector
 second
 set!
 set-time-nanosecond!
 set-time-second!
 set-time-type!
 seventh
 shape
 share-array
 sixth
 span
 span!
 split-at
 split-at!
SRFI

SRFI 11: Syntax for receiving multiple values

SRFI 13: String Libraries

SRFI 14: Character-set Library

SRFI 16: Syntax for procedures of variable arity

SRFI 17: Generalized set!

SRFI 19: Time Data Types and Procedures

SRFI 1: List Library

SRFI 23: Error reporting mechanism	values
SRFI 25: Multi-dimensional Array Primitives	SRFI 74: Octet-Addressed Binary Blocks
SRFI 26: Notation for Specializing Parameters without Currying	SRFI 78: Lightweight testing
SRFI 27: Sources of Random Bits	SRFI 7: Feature-based program configuration language
SRFI 28: Basic Format Strings	SRFI 86: MU & NU simulating VALUES & CALL-WITH-VALUES...
SRFI 29: Localization	SRFI 87: => in case clauses
SRFI 2: AND-LET*: an AND with local bindings...	SRFI 8: RECEIVE: Binding to multiple values
SRFI 30: Nested Multi-line Comments	SRFI 98: An interface to access environment variables
SRFI 31: A special form rec for recursive evaluation	SRFI 9: Defining Record Types
SRFI 34: Exception Handling for Programs	srfi/1
SRFI 35: Conditions	srfi/11
SRFI 38: External Representation for Data With Shared Structure	srfi/13
SRFI 39: Parameter objects	srfi/14
SRFI 40: A Library of Streams	srfi/16
SRFI 41: Streams	srfi/17
SRFI 42: Eager Comprehensions	srfi/19
SRFI 43: Vector Library	srfi/2
SRFI 45: Primitives for Expressing Iterative Lazy Algorithms	srfi/23
SRFI 48: Intermediate Format Strings	srfi/25
SRFI 4: Homogeneous numeric vector datatypes	srfi/26
SRFI 54: Formatting	srfi/27
SRFI 57: Records	srfi/28
SRFI 59: Vicinity	srfi/29
SRFI 5: A compatible let form with signatures and rest arguments	srfi/30
SRFI 60: Integers as Bits	srfi/31
SRFI 61: A more general cond clause	srfi/34
SRFI 62: S-expression comments	srfi/35
SRFI 63: Homogeneous and Heterogeneous Arrays	srfi/38
SRFI 64: A Scheme API for test suites	srfi/39
SRFI 66: Octet Vectors	srfi/4
SRFI 67: Compare Procedures	srfi/40
SRFI 69: Basic hash tables	srfi/41
SRFI 6: Basic String Ports	srfi/42
SRFI 71: Extended LET-syntax for multiple	srfi/43
	srfi/45
	srfi/48
	srfi/5
	srfi/54

srfi/57	stream-iterate
srfi/59	stream-lambda
srfi/6	stream-length
srfi/60	stream-let
srfi/61	stream-map
srfi/63	stream-map
srfi/64	stream-match
srfi/66	stream-null
srfi/67	stream-null
srfi/69	stream-null?
srfi/7	stream-null?
srfi/71	stream-of
srfi/74	stream-pair?
srfi/78	stream-range
srfi/8	stream-ref
srfi/86	stream-reverse
srfi/87	stream-scan
srfi/9	stream-take
srfi/98	stream-take-while
SRFIs: Libraries	stream-unfold
store-bundle	stream-unfoldn
stream	stream-zip
stream	stream?
stream->list	stream?
stream-append	string
stream-car	string->char-set
stream-car	string->char-set!
stream-cdr	string->date
stream-cdr	string->list
stream-concat	string-any
stream-cons	string-append
stream-cons	string-append-ec
stream-constant	string-append/shared
stream-delay	string-ci<
stream-drop	string-ci<=
stream-drop-while	string-ci<>
stream-filter	string-ci=
stream-filter	string-ci>
stream-fold	string-ci>=
stream-for-each	string-compare
stream-for-each	string-compare-ci
stream-from	string-concatenate

string-concatenate-reverse	string-reverse!
string-concatenate-reverse/shared	string-set!
string-concatenate/shared	string-skip
string-contains	string-skip-right
string-contains-ci	string-suffix-ci?
string-copy	string-suffix-length
string-copy!	string-suffix-length-ci
string-count	string-suffix?
string-delete	string-tabulate
string-downcase	string-take
string-downcase!	string-take-right
string-drop	string-titlecase
string-drop-right	string-titlecase!
string-ec	string-tokenize
string-every	string-trim
string-fill!	string-trim-both
string-filter	string-trim-right
string-fold	string-unfold
string-fold-right	string-unfold-right
string-for-each	string-upcase
string-for-each-index	string-upcase!
string-hash	string-xcopy!
string-hash-ci	string<
string-index	string<=
string-index-right	string<>
string-join	string=
string-kmp-partial-search	string>
string-length	string>=
string-map	string?
string-map!	sub-vicinity
string-null?	substring-spec-ok?
string-pad	substring/shared
string-pad-right	subtract-duration
string-parse-final-start+end	subtract-duration!
string-parse-start+end	sum-ec
string-prefix-ci?	take
string-prefix-length	take!
string-prefix-length-ci	take-right
string-prefix?	take-while
string-ref	take-while!
string-replace	tenth
string-reverse	third

time-difference	u64vector
time-difference!	u8vector
time-duration	ucs-range->char-set
time-monotonic	ucs-range->char-set!
time-monotonic->date	unfold
time-monotonic->julian-day	unfold-right
time-monotonic->modified-julian-day	unzip1
	unzip2
time-monotonic->time-tai	unzip3
time-monotonic->time-tai!	unzip4
time-monotonic->time-utc	unzip5
time-monotonic->time-utc!	user-vicinity
time-nanosecond	vector
time-process	vector->list
time-resolution	vector-any
time-second	vector-append
time-tai	vector-binary-search
time-tai->date	vector-concatenate
time-tai->julian-day	vector-copy
time-tai->modified-julian-day	vector-copy!
time-tai->time-monotonic	vector-count
time-tai->time-monotonic!	vector-ec
time-tai->time-utc	vector-empty?
time-tai->time-utc!	vector-every
time-thread	vector-fill!
time-type	vector-fold
time-utc	vector-fold-right
time-utc->date	vector-for-each
time-utc->julian-day	vector-index
time-utc->modified-julian-day	vector-index-right
time-utc->time-monotonic	vector-length
time-utc->time-monotonic!	vector-map
time-utc->time-tai	vector-map!
time-utc->time-tai!	vector-of-length-ec
time<=?	vector-ref
time<?	vector-reverse!
time=?	vector-reverse-copy
time>=?	vector-reverse-copy!
time>?	vector-set!
time?	vector-skip
u16vector	vector-skip-right
u32vector	vector-swap!

vector-unfold
vector-unfold-right
vector=
vector?
vicinity:suffix?
with-exception-handler
write-with-shared-structure
xcons
xsubstring
zip