

guile-r7rs

[sourcehut](#) [success](#)

Introduction

guile-r7rs is the collection of libraries part of R7RS bundled for GNU Guile 2.2 or later.

How to contribute

1. Create an account on sr.ht. To contribute to existing repository, it is free.
2. Pick an item and check nobody is working on it in the todo.
3. Add documentation, tests or an implementation based on existing Guile modules or sample implementation that can be found at <http://srfi.schemers.org/>. Also, R7RS small is implemented in terms of R6RS in akku-r7rs in a compatible license.
4. When your contribution is ready, ask amirouche at hyper dev to become a contributor to be able to push.

Don't forget to add your name in the license header.

When you add a documentation file, don't forget to add it to DOCUMENTATION_FILES inside the `Makefile`. To build the documentation you will need `pandoc`, `latex` and to run `make doc`.

To run the tests use `make check`.

Table of Content

R7RS small

- (`(scheme base)`)
- (`(scheme case-lambda)`)
- (`(scheme char)`)
- (`(scheme complex)`)
- (`(scheme cxr)`)
- (`(scheme eval)`)
- (`(scheme file)`)
- (`(scheme inexact)`)
- (`(scheme lazy)`)
- (`(scheme load)`)

- `(scheme process-context)`
- `(scheme r5rs)`
- `(scheme read)`
- `(scheme repl)`
- `(scheme time)`
- `(scheme write)`

R7RS Red Edition

- `(scheme box)` aka. SRFI 111
- `(scheme charset)` aka. SRFI 14
- `(scheme comparator)` aka. SRFI 128
- `(scheme ephemerons)` aka. SRFI 124
- `(scheme hash-table)` aka. SRFI 125
- `(scheme ideque)` aka. SRFI 134
- `(scheme ilist)` aka. SRFI 116
- `(scheme list)` aka. SRFI 1
- `(scheme list-queue)` aka. SRFI 117
- `(scheme lseq)` aka. SRFI 127
- `(scheme rlist)` aka SRFI 101
- `(scheme set)` aka. SRFI 113
- `(scheme sort)` aka. SRFI 132
- `(scheme stream)` aka. SRFI 41
- `(scheme text)` aka. SRFI 135
- `(scheme vector)` aka. SRFI 133

R7RS Tangerine Edition

- `(scheme mapping)` aka. SRFI 146
- `(scheme mapping hash)` aka. SRFI 146
- `(scheme regex)` aka. SRFI 115
- `(scheme generator)` aka. SRFI 158
- `(scheme division)` aka. SRFI 141
- `(scheme bitwise)` aka. SRFI 151
- `(scheme fixnum)` aka. SRFI 143
- `(scheme flonum)` aka. SRFI 144
- `(scheme bytvector)` aka. `(rnrs bytectors)` aka. SRFI 4
- `(scheme vector @)` aka. SRFI 160 where @ is any of base, u8, s8, u16, s16, u32, s32, u64, s64, f32, f64, c64, cl28.
- `(scheme show)` aka. SRFI 159

(scheme base)

-

TODO (missing in r7rs?)

...

It is called ellipsis. It used in macros, `match` that is not part of R7RS. It signify that a pattern must be repeated.

=>

TODO

else

Used in `cond` form as in the last clause as a fallback.

(* number ...)

Multiplication procedure.

(+ number ...)

Addition procedure.

(- number ...)

Substraction procedure.

(/ number number ...)

Division procedure. Raise '`numerical-overflow`' condition in case where denominator is zero.

(< number number ...)

Less than procedure. Return a boolean.

(<= number number ...)

Less than or equal procedure. Return a boolean.

(= number number ...)

Return #t if the numbers passed as parameters are equal. And #f otherwise.

(> number number ...)

Greater than procedure. Return a boolean.

(>= number number ...)

Greater than or equal. Return a boolean.

(abs number)

Return the absolute value of NUMBER.

(and test1 ...)

The **test** expressions are evaluated from left to right, and if any expression evaluates to #f, then #f is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the values of the last expression are returned. If there are no expressions, then #t is returned.

(append lst ...)

Return the list made of the list passed as parameters in the same order.

(apply proc arg1 ... args)

The apply procedure calls proc with the elements of the list (append (list arg1 ...) args) as the actual arguments.

(assoc obj alist)

Return the first pair which **car** is equal to OBJ according to the predicate **equal?**. Or it returns #f.

(assq obj alist)

Return the first pair which `car` is equal to `OBJ` according to the predicate `eq?`.
Or it returns `#f`.

(assv obj alist)

Return the first pair which `car` is equal to `OBJ` according to the predicate `eqv?`.
Or it returns `#f`.

begin syntax

There are two uses of `begin`.

(begin expression-or-definition ...)

This form of `begin` can appear as part of a body, or at the outermost level of a program, or at the REPL, or directly nested in a `begin` that is itself of this form. It causes the contained expressions and definitions to be evaluated exactly as if the enclosing `begin` construct were not present.

TODO: example

(begin expression1 expression2 ...)

This form of `begin` can be used as an ordinary expression. The expressions are evaluated sequentially from left to right, and the values of the last expression are returned. This expression type is used to sequence side effects such as assignments or input and output.

TODO: example

binary-port?

TODO: not implemented

(boolean=? obj ...)

Return `#t` if the scheme objects passed as arguments are the same boolean.
Otherwise it returns `#f`.

(boolean? obj)

Return #t if OBJ is a boolean. Otherwise #f.

(bytevector byte ...)

Returns a newly allocated bytevector containing its arguments.

(bytevector-append bytevector ...)

Returns a newly allocated bytevector whose elements are the concatenation of the elements in the given bytevectors.

(bytevector-copy bytevector [start [end]])

Returns a newly allocated bytevector containing the bytes in bytevector between start and end.

(bytevector-copy! to at from [start [end]])

Copies the bytes of bytevector from between start and end to bytevector TO, starting at at. The order in which bytes are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary bytevector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

(bytevector-length bytevector)

Returns the length of bytevector in bytes as an exact integer.

bytevector-u8-ref

Returns the Kth byte of BYTEVECTOR. It is an error if K is not a valid index of BYTEVECTOR.

bytevector-u8-set!

Stores BYTE as the Kth byte of BYTEVECTOR.

It is an error if K is not a valid index of BYTEVECTOR.

(bytevector? obj)

Returns #t if OBJ is a bytevector. Otherwise, #f is returned.

caar

TODO

cadr

TODO

(call-with-current-continuation proc)

It is an error if proc does not accept one argument.

The procedure call-with-current-continuation (or its equivalent abbreviation call/cc) packages the current continuation (see the rationale below) as an “escape procedure” and passes it as an argument to proc. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. Calling the escape procedure will cause the invocation of before and after thunks installed using dynamic-wind.

The escape procedure accepts the same number of arguments as the continuation to the original call to call-with-current-continuation. Most continuations take only one value. Continuations created by the call-with-values procedure (including the initialization expressions of define-values, let-values, and let-values expressions), *take the number of values that the consumer expects. The continuations of all non-final expressions within a sequence of expressions, such as in lambda, case-lambda, begin, let, let, letrec, letrec, let-values, let-values, let-syntax, letrec-syntax, parameterize, guard, case, cond, when, and unless expressions, take an arbitrary number of values because they discard the values passed to them in any event.* The effect of passing no values or more than one value to continuations that were not created in one of these ways is unspecified.

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It can be stored in variables or data structures and can be called as many times as desired. However, like the raise and error procedures, it never returns to its caller.

TODO: example

(call-with-port port proc)

The `call-with-port` procedure calls `PROC` with `PORT` as an argument. If `PROC` returns, then the `PORT` is closed automatically and the values yielded by the `PROC` are returned. If `PROC` does not return, then the `PORT` must not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

It is an error if `PROC` does not accept one argument.

(call-with-values producer consumer)

Calls its producer argument with no arguments and a continuation that, when passed some values, calls the consumer procedure with those values as arguments. The continuation for the call to consumer is the continuation of the call to `call-with-values`.

call/cc

Abbreviation for `call-with-continuation`.

car

Returns the contents of the `car` field of pair. Note that it is an error to take the `car` of the empty list.

(case key clause1 clause2 ...) syntax

TODO

cdar

TODO

cddr

TODO

cdr

Returns the contents of the `cdr` field of pair. Note that it is an error to take the `cdr` of the empty list.

(ceiling x)

The ceiling procedure returns the smallest integer not smaller than x.

(char->integer char)

Given a Unicode character, **char->integer** returns an exact integer between 0 and #xD7FF or between #xE000 and #x10FFFF which is equal to the Unicode scalar value of that character. Given a non-Unicode character, it returns an exact integer greater than #x10FFFF.

(char-ready? [port])

Returns #t if a character is ready on the textual input port and returns #f otherwise. If char-ready returns #t then the next read-char operation on the given port is guaranteed not to hang. If the port is at end of file then char-ready? returns #t.

char<=?

TODO

char<?

TODO

char=?

TODO

char>=?

TODO

char>?

TODO

char?

Returns #t if obj is a character, otherwise returns #f.

(close-input-port port)

Closes the resource associated with port, rendering the port incapable of delivering or accepting data.

(close-output-port port)

Closes the resource associated with port, rendering the port incapable of delivering or accepting data.

(close-port port)

Closes the resource associated with port, rendering the port incapable of delivering or accepting data.

complex?

Returns #t if obj is a complex number, otherwise returns #f.

cond

TODO

cond-expand

TODO: not implemented

(cons obj1 obj2)

Returns a newly allocated pair whose car is obj1 and whose cdr is obj2. The pair is guaranteed to be different (in the sense of eqv?) from every existing object.

current-error-port

Returns the current default error port (an output port). That procedure is also a parameter object, which can be overridden with `parameterize`.

current-input-port

Returns the current default input port. That procedure is also a parameter object, which can be overridden with `parameterize`.

current-output-port

Returns the current default output port. That procedure is also a parameter object, which can be overridden with `parameterize`.

define

TODO

define-record-type

TODO

define-syntax

TODO

(define-values var1 ... expr) syntax

creates multiple definitions from a single expression returning multiple values. It is allowed wherever `define` is allowed.

(denominator q)

Return the denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

do

TODO

(dynamic-wind before thunk after)

TODO

(eof-object)

Returns an end-of-file object, not necessarily unique.

(eof-object? obj)

Returns #t if obj is an end-of-file object, otherwise returns #f. A end-of-file object will ever be an object that can be read in using read.

(eq? obj1 obj2)

The eq? procedure is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?. It must always return #f when eqv? also would, but may return #f in some cases where eqv? would return #t.

On symbols, booleans, the empty list, pairs, and records, and also on non-empty strings, vectors, and bytevectors, eq? and eqv? are guaranteed to have the same behavior. On procedures, eq? must return true if the arguments' location tags are equal. On numbers and characters, eq?'s behavior is implementation-dependent, but it will always return either true or false. On empty strings, empty vectors, and empty bytevectors, eq? may also behave differently from eqv?.

(equal? obj1 obj2)

The equal? procedure, when applied to pairs, vectors, strings and bytevectors, recursively compares them, returning #t when the unfoldings of its arguments into (possibly infinite) trees are equal (in the sense of equal?) as ordered trees, and #f otherwise. It returns the same as eqv? when applied to booleans, symbols, numbers, characters, ports, procedures, and the empty list. If two objects are eqv?, they must be equal? as well. In all other cases, equal? may return either #t or #f.

Even if its arguments are circular data structures, equal? must always terminate.

(eqv? obj1 obj2)

The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 are normally regarded as the same object.

TODO: complete based on r7rs small and guile.

```
(error [who] message . irritants)
```

Raises an exception as if by calling raise on a newly allocated implementation-defined object which encapsulates the information provided by message, as well as any objs, known as the irritants. The procedure error-object? must return #t on such objects.

```
(error-object-irritants error)
```

Returns a list of the irritants encapsulated by error.

```
(error-object-message error)
```

Returns the message encapsulated by error.

```
(error-object? obj)
```

Returns #t if obj is an object created by `error` or one of an implementation-defined set of objects. Otherwise, it returns #f. The objects used to signal errors, including those which satisfy the predicates `file-error?` and `read-error?`, may or may not satisfy `error-object?`.

```
(even? number)
```

Return #t if NUMBER is even. Otherwise #f.

```
(exact z)
```

TODO: FIXME

The procedure exact returns an exact representation of z. The value returned is the exact number that is numerically closest to the argument. For exact arguments, the result is the same as the argument. For inexact non-integral real arguments, the implementation may return a rational approximation, or may report an implementation violation. For inexact complex arguments, the result is a complex number whose real and imaginary parts are the result of applying exact to the real and imaginary parts of the argument, respectively. If an inexact argument has no reasonably close exact equivalent, (in the sense of =), then a violation of an implementation restriction may be reported.

`(exact-integer-sqrt k)`

TODO

`(exact-integer? z)`

Returns #t if z is both exact and an integer; otherwise returns #f.

`(exact? z)`

Return #t if Z is exact. Otherwise #f.

`(expt z1 z2)`

Returns z1 raised to the power z2.

features

TODO: no implemented

`(file-error? error)`

TODO: not implemented?

`(floor x)`

The floor procedure returns the largest integer not larger than x.

floor-quotient

TODO

floor-remainder

TODO

`floor/`

TODO

(flush-output-port [port])

Flushes any buffered output from the buffer of output-port to the underlying file or device and returns an unspecified value.

(for-each proc list1 ...)

It is an error if proc does not accept as many arguments as there are lists.

The arguments to for-each are like the arguments to map, but for-each calls proc for its side effects rather than for its values. Unlike map, for-each is guaranteed to call proc on the elements of the lists in order from the first element(s) to the last, and the value returned by for-each is unspecified. If more than one list is given and not all lists have the same length, for-each terminates when the shortest list runs out. The lists can be circular, but it is an error if all of them are circular.

(gcd n1 ...)

Return the greatest common divisor.

get-output-bytevector

It is an error if port was not created with open-output-bytevector.

Returns a bytevector consisting of the bytes that have been output to the port so far in the order they were output.

(get-output-string port)

It is an error if port was not created with open-output-string.

Returns a string consisting of the characters that have been output to the port so far in the order they were output.

guard

TODO

(if expr then [else])

TODO

include

TODO: not implemented?

include-ci

TODO: not implemented?

(inexact z)

The procedure `inexact` returns an inexact representation of `z`. The value returned is the inexact number that is numerically closest to the argument. For inexact arguments, the result is the same as the argument. For exact complex numbers, the result is a complex number whose real and imaginary parts are the result of applying `inexact` to the real and imaginary parts of the argument, respectively. If an exact argument has no reasonably close inexact equivalent (in the sense of `=`), then a violation of an implementation restriction may be reported.

(inexact? z)

Return #t if `Z` is inexact. Otherwise #f.

(input-port-open? port)

Returns #t if `port` is still open and capable of performing input, and #f otherwise.

(input-port? obj)

Return #t if `obj` is an input port. Otherwise it return #f.

(integer->char integer)

Given an exact integer that is the value returned by a character when `char->integer` is applied to it, `integer->char` returns that character.

(integer? obj)

Return #t if `OBJ` is an integer. Otherwise #f.

lambda

TODO

(lcm n1 ...)

Return the least common multiple of its arguments.

(length list)

Returns the length of list.

let

TODO

let*

TODO

let*-values

TODO

let-syntax

TODO

let-values

TODO

letrec

TODO

letrec*

TODO

letrec-syntax

TODO

(list obj ...)

Returns a newly allocated list of its arguments.

(list->string list)

It is an error if any element of list is not a character.

list->string returns a newly allocated string formed from the elements in the list list.

(list->vector list)

The list->vector procedure returns a newly created vector initialized to the elements of the list list.

(list-copy obj)

Returns a newly allocated copy of the given obj if it is a list. Only the pairs themselves are copied; the cars of the result are the same (in the sense of eqv?) as the cars of list. If obj is an improper list, so is the result, and the final cdrs are the same in the sense of eqv?. An obj which is not a list is returned unchanged. It is an error if obj is a circular list.

(list-ref list k)

The list argument can be circular, but it is an error if list has fewer than k elements.

Returns the kth element of list. (This is the same as the car of (list-tail list k).)

(list-set! list k obj)

It is an error if k is not a valid index of list.

The list-set! procedure stores obj in element k of list.

(list-tail list k)

It is an error if list has fewer than k elements.

Returns the sublist of list obtained by omitting the first k elements.

(list? obj)

Return #t if OBJ is a list. Otherwise #f.

(make-bytevector k [byte])

The make-bytevector procedure returns a newly allocated bytevector of length k. If byte is given, then all elements of the bytevector are initialized to byte, otherwise the contents of each element are unspecified.

(make-list k [fill])

Returns a newly allocated list of k elements. If a second argument is given, then each element is initialized to fill. Otherwise the initial contents of each element is unspecified.

(make-parameter init [converter])

Returns a newly allocated parameter object, which is a procedure that accepts zero arguments and returns the value associated with the parameter object. Initially, this value is the value of (converter init), or of init if the conversion procedure converter is not specified. The associated value can be temporarily changed using parameterize, which is described below.

(make-string k [char])

The make-string procedure returns a newly allocated string of length k. If char is given, then all the characters of the string are initialized to char, otherwise the contents of the string are unspecified.

(make-vector k [fill])

Returns a newly allocated vector of k elements. If a second argument is given, then each element is initialized to fill. Otherwise the initial contents of each element is unspecified.

(map proc list1 ...)

It is an error if proc does not accept as many arguments as there are lists and return a single value.

The map procedure applies proc element-wise to the elements of the lists and returns a list of the results, in order. If more than one list is given and not all lists have the same length, map terminates when the shortest list runs out. The lists can be circular, but it is an error if all of them are circular. It is an error for proc to mutate any of the lists. The dynamic order in which proc is applied to the elements of the lists is unspecified. If multiple returns occur from map, the values returned by earlier returns are not mutated.

(max x1 ...)

Return the maximum of its arguments.

(member obj list [compare])

Return the first sublist of list whose `car` is `obj`, where the sublists of list are the non-empty lists returned by (list-tail list k) for k less than the length of list. If `obj` does not occur in `list`, then `#f` (not the empty list) is returned.

Uses `compare`, if given, and `equal?` otherwise.

(memq obj list)

Return the first sublist of list whose `car` is `obj`, where the sublists of list are the non-empty lists returned by (list-tail list k) for k less than the length of list. If `obj` does not occur in `list`, then `#f` (not the empty list) is returned.

Use `eq?` for comparison.

(memv obj list)

Return the first sublist of list whose `car` is `obj`, where the sublists of list are the non-empty lists returned by (list-tail list k) for k less than the length of list. If `obj` does not occur in `list`, then `#f` (not the empty list) is returned.

Uses `eqv?` for comparison.

(min x1 ...)

Return the minimum of its arguments.

(modulo n1 n2)

modulo is equivalent to **floor-remainder**. Provided for backward compatibility.

(negative? x)

Return #t if X is negative. Otherwise #f.

(newline [port])

Writes an end of line to output port.

(not obj)

The not procedure returns #t if obj is false, and returns #f otherwise.

(null? obj)

Returns #t if obj is the empty list, otherwise returns #f.

(number->string z [radix])

It is an error if radix is not one of 2, 8, 10, or 16.

(number? obj)

Return #t if OBJ is a number. Otherwise #f.

(numerator q)

TODO

(odd? number)

Return #t if NUMBER is odd. Otherwise #f.

(open-input-bytevector bytevector)

Takes a bytevector and returns a binary input port that delivers bytes from the bytevector.

(open-input-string string)

Takes a string and returns a textual input port that delivers characters from the string. If the string is modified, the effect is unspecified.

(open-output-bytevector)

Returns a binary output port that will accumulate bytes for retrieval by `get-output-bytevector`.

(open-output-string)

Returns a textual output port that will accumulate characters for retrieval by `get-output-string`.

(or test1 ...) *syntax*

The `test` expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to #f or if there are no expressions, then #f is returned.

(output-port-open? port)

Returns #t if port is still open and capable of performing output, and #f otherwise.

(output-port? obj)

Return #t if obj is an output port. Otherwise return #f.

(pair? obj)

The pair? predicate returns #t if obj is a pair, and otherwise returns #f.

(parameterize ((param1 value1) ...) expr ...)

A parameterize expression is used to change the values returned by specified parameter objects during the evaluation of the body.

The param and value expressions are evaluated in an unspecified order. The body is evaluated in a dynamic environment in which calls to the parameters return the results of passing the corresponding values to the conversion procedure specified when the parameters were created. Then the previous values of the parameters are restored without passing them to the conversion procedure. The results of the last expression in the body are returned as the results of the entire parameterize expression.

Note: If the conversion procedure is not idempotent, the results of (parameterize ((x (x)) ...)), which appears to bind the parameter x to its current value, might not be what the user expects.

If an implementation supports multiple threads of execution, then parameterize must not change the associated values of any parameters in any thread other than the current thread and threads created inside body.

Parameter objects can be used to specify configurable settings for a computation without the need to pass the value to every procedure in the call chain explicitly.

(**peek-char** [port])

Returns the next character available from the textual input port, but without updating the port to point to the following character. If no more characters are available, an end-of-file object is returned.

Note: The value returned by a call to peek-char is the same as the value that would have been returned by a call to read-char with the same port. The only difference is that the very next call to read-char or peek-char on that port will return the value returned by the preceding call to peek-char. In particular, a call to peek-char on an interactive port will hang waiting for input whenever a call to read-char would have hung.

(**peek-u8** [port])

Returns the next byte available from the binary input port, but without updating the port to point to the following byte. If no more bytes are available, an end-of-file object is returned.

(**port?** obj)

Return #t if OBJ is port. Otherwise #f.

(**positive?** x)

Return #t if X is positive. Otherwise #f.

(procedure? obj)

Return #t if OBJ is a procedure. Otherwise #f.

quasiquote

TODO

quote

TODO

quotient

TODO

(raise obj)

Raises an exception by invoking the current exception handler on obj. The handler is called with the same dynamic environment as that of the call to raise, except that the current exception handler is the one that was in place when the handler being called was installed. If the handler returns, a secondary exception is raised in the same dynamic environment as the handler. The relationship between obj and the object raised by the secondary exception is unspecified.

(raise-continuable obj)

Raises an exception by invoking the current exception handler on obj. The handler is called with the same dynamic environment as the call to raise-continuable, except that: (1) the current exception handler is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to raise-continuable.

(rational? obj)

Return #t if OBJ is a rational number. Otherwise #f.

```
(rationalize x y)
```

The rationalize procedure returns the simplest rational number differing from x by no more than y.

```
(read-bytevector k [port])
```

Reads the next k bytes, or as many as are available before the end of file, from the binary input port into a newly allocated bytevector in left-to-right order and returns the bytevector. If no bytes are available before the end of file, an end-of-file object is returned.

```
(read-bytevector! bytevector [port [start [end]]])
```

Reads the next end - start bytes, or as many as are available before the end of file, from the binary input port into bytevector in left-to-right order beginning at the start position. If end is not supplied, reads until the end of bytevector has been reached. If start is not supplied, reads beginning at position 0. Returns the number of bytes read. If no bytes are available, an end-of-file object is returned.

```
(read-char [port])
```

Returns the next character available from the textual input port, updating the port to point to the following character. If no more characters are available, an end-of-file object is returned.

```
(read-error? obj)
```

Error type predicates. Returns #t if obj is an object raised by the read procedure. Otherwise, it returns #f.

```
(read-line [port])
```

Returns the next line of text available from the textual input port, updating the port to point to the following character. If an end of line is read, a string containing all of the text up to (but not including) the end of line is returned, and the port is updated to point just past the end of line. If an end of file is encountered before any end of line is read, but some characters have been read, a string containing those characters is returned. If an end of file is encountered before any characters are read, an end-of-file object is returned. For the purpose of this procedure, an end of line consists of either a linefeed character, a carriage return character, or a sequence of a carriage return character followed by a linefeed

character. Implementations may also recognize other end of line characters or sequences.

(read-string k [port])

Reads the next k characters, or as many as are available before the end of file, from the textual input port into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

(read-u8 [port])

Returns the next byte available from the binary input port, updating the port to point to the following byte. If no more bytes are available, an end-of-file object is returned.

(real? obj)

Return #t if OBJ is real number. Otherwise #f.

(remainder n1 n2)

TODO

(reverse list)

Returns a newly allocated list consisting of the elements of list in reverse order.

(round x)

TODO

(set! variable expression) syntax

Expression is evaluated, and the resulting value is stored in the location to which variable is bound. It is an error if variable is not bound either in some region enclosing the set! expression or else globally. The result of the set! expression is unspecified.

(set-car! pair obj)

Stores obj in the car field of pair.

(set-cdr! pair obj)

Stores obj in the cdr field of pair.

(square z)

Returns the square of z. This is equivalent to (* z z).

(string char ...)

Returns a newly allocated string composed of the arguments. It is analogous to list.

(string->list string [start [end]])

The string->list procedure returns a newly allocated list of the characters of string between start and end.

(string->number string [radix])

Returns a number of the maximally precise representation expressed by the given string. It is an error if radix is not 2, 8, 10, or 16.

If supplied, radix is a default radix that will be overridden if an explicit radix prefix is present in string (e.g. "#o177"). If radix is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number, or would result in a number that the implementation cannot represent, then string->number returns #f. An error is never signaled due to the content of string.

(string->symbol string)

Returns the symbol whose name is string. This procedure can create symbols with names containing special characters that would require escaping when written, but does not interpret escapes in its input.

```
(string->utf8 string [start [end]])
```

The string->utf8 procedure encodes the characters of a string between start and end and returns the corresponding bytevector.

```
(string->vector string [start [end]])
```

The string->vector procedure returns a newly created vector initialized to the elements of the string string between start and end.

```
(string-append string ...)
```

Returns a newly allocated string whose characters are the concatenation of the characters in the given strings.

```
(string-copy string [start [end]])
```

Returns a newly allocated copy of the part of the given string between start and end.

```
(string-copy! to at from [start [end]])
```

It is an error if at is less than zero or greater than the length of to. It is also an error if (- (string-length to) at) is less than (- end start).

Copies the characters of string from between start and end to string to, starting at at. The order in which characters are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

```
(string-fill! string fill [start [end]])
```

It is an error if fill is not a character.

The string-fill! procedure stores fill in the elements of string between start and end.

(string-for-each proc string1 ...)

It is an error if proc does not accept as many arguments as there are strings.

The arguments to string-for-each are like the arguments to string-map, but string-for-each calls proc for its side effects rather than for its values. Unlike string-map, string-for-each is guaranteed to call proc on the elements of the lists in order from the first element(s) to the last, and the value returned by string-for-each is unspecified. If more than one string is given and not all strings have the same length, string-for-each terminates when the shortest string runs out. It is an error for proc to mutate any of the strings.

(string-length string)

Returns the number of characters in the given string.

(string-map proc string1 ...)

It is an error if proc does not accept as many arguments as there are strings and return a single character.

The string-map procedure applies proc element-wise to the elements of the strings and returns a string of the results, in order. If more than one string is given and not all strings have the same length, string-map terminates when the shortest string runs out. The dynamic order in which proc is applied to the elements of the strings is unspecified. If multiple returns occur from string-map, the values returned by earlier returns are not mutated.

(string-ref string k)

It is an error if k is not a valid index of string.

The string-ref procedure returns character k of string using zero-origin indexing. There is no requirement for this procedure to execute in constant time.

(string-set! string k char)

It is an error if k is not a valid index of string.

The string-set! procedure stores char in element k of string. There is no requirement for this procedure to execute in constant time.

string<=?

TODO

string<?

TODO

(string=? string1 string2 ...)

Returns #t if all the strings are the same length and contain exactly the same characters in the same positions, otherwise returns #f.

string>=?

TODO

string>?

TODO

(string? obj)

Return #t if OBJ is string. Otherwise #f.

(substring string start end)

The substring procedure returns a newly allocated string formed from the characters of string beginning with index start and ending with index end. This is equivalent to calling string-copy with the same arguments, but is provided for backward compatibility and stylistic flexibility.

(symbol->string symbol)

Returns the name of symbol as a string, but without adding escapes. It is an error to apply mutation procedures like string-set! to strings returned by this procedure.

`(symbol=? symbol1 symbol2 ...)`

Returns #t if all the arguments are symbols and all have the same names in the sense of string=?.

`(symbol? obj)`

Returns #t if obj is a symbol, otherwise returns #f.

`syntax-error`

TODO

`syntax-rules`

TODO

`textual-port?`

TODO

`(truncate x)`

TODO

`truncate-quotient`

TODO

`truncate-remainder`

TODO

`truncate/`

TODO

(u8-ready? [port])

Returns #t if a byte is ready on the binary input port and returns #f otherwise. If u8-ready? returns #t then the next read-u8 operation on the given port is guaranteed not to hang. If the port is at end of file then u8-ready? returns #t.

(unless test expr ...) syntax

The test is evaluated, and if it evaluates to #f, the expressions are evaluated in order. The result of the unless expression is unspecified.

unquote

TODO

unquote-splicing

TODO

(utf8->string bytevector [start [end]])

It is an error for bytevector to contain invalid UTF-8 byte sequences.

The utf8->string procedure decodes the bytes of a bytevector between start and end and returns the corresponding string

(values obj ...)

Delivers all of its arguments to its continuation.

(vector obj ...)

Returns a newly allocated vector whose elements contain the given arguments. It is analogous to list.

(vector->list vector [start [end]])

The vector->list procedure returns a newly allocated list of the objects contained in the elements of vector between start and end. The list->vector procedure returns a newly created vector initialized to the elements of the list list.

```
(vector->string vector [start [end]])
```

It is an error if any element of vector between start and end is not a character.

The vector->string procedure returns a newly allocated string of the objects contained in the elements of vector between start and end. The string->vector procedure returns a newly created vector initialized to the elements of the string string between start and end.

```
(vector-append vector ...)
```

Returns a newly allocated vector whose elements are the concatenation of the elements of the given vectors.

```
(vector-copy vector [start [end]])
```

Returns a newly allocated copy of the elements of the given vector between start and end. The elements of the new vector are the same (in the sense of eqv?) as the elements of the old.

```
(vector-copy! to at from [start [end]])
```

It is an error if at is less than zero or greater than the length of to. It is also an error if (- (vector-length to) at) is less than (- end start).

Copies the elements of vector from between start and end to vector to, starting at at. The order in which elements are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary vector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

```
(vector-fill! vector fill [start [end]])
```

The vector-fill! procedure stores fill in the elements of vector between start and end.

```
(vector-for-each proc vector1 ...)
```

It is an error if proc does not accept as many arguments as there are vectors.

The arguments to vector-for-each are like the arguments to vector-map, but vector-for-each calls proc for its side effects rather than for its values. Unlike vector-map, vector-for-each is guaranteed to call proc on the elements of the

vectors in order from the first element(s) to the last, and the value returned by vector-for-each is unspecified. If more than one vector is given and not all vectors have the same length, vector-for-each terminates when the shortest vector runs out. It is an error for proc to mutate any of the vectors.

(vector-length vector)

Returns the number of elements in vector as an exact integer.

(vector-map proc vector1 ...)

It is an error if proc does not accept as many arguments as there are vectors and return a single value.

The vector-map procedure applies proc element-wise to the elements of the vectors and returns a vector of the results, in order. If more than one vector is given and not all vectors have the same length, vector-map terminates when the shortest vector runs out. The dynamic order in which proc is applied to the elements of the vectors is unspecified. If multiple returns occur from vector-map, the values returned by earlier returns are not mutated.

(vector-ref vector k)

It is an error if k is not a valid index of vector.

The vector-ref procedure returns the contents of element k of vector.

(vector-set! vector k obj)

It is an error if k is not a valid index of vector.

The vector-set! procedure stores obj in element k of vector.

vector?

Returns #t if obj is a bytevector. Otherwise, #f is returned.

(when test expr ...) syntax

The test is evaluated, and if it evaluates to a true value, the expressions are evaluated in order. The result of the when expression is unspecified.

`with-exception-handler`

TODO

`(write-bytevector bytevector [port [start [end]]])`

Writes the bytes of bytevector from start to end in left-to-right order to the binary output port.

`(write-char char [port])`

Writes the character char (not an external representation of the character) to the given textual output port and returns an unspecified value.

`(write-string string [port [start [end]]])`

Writes the characters of string from start to end in left-to-right order to the textual output port.

`(write-u8 byte [port])`

Writes the byte to the given binary output port and returns an unspecified value.

`(zero? z)`

Return #t if z is zero. Otherwise #f.

`(scheme case-lambda)`

`(case-lambda clause1 clause2 ...) syntax`

Each clause is of the form `(formals body)`, where `formals` and `body` have the same syntax as in a lambda expression.

A case-lambda expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as a procedure resulting from a lambda expression. When the procedure is called, the first clause for which the arguments agree with `formals` is selected, where agreement is specified as for the `formals` of a lambda expression. The variables of `formals` are bound to fresh locations, the values of the arguments are stored in those locations, the `body` is evaluated in the extended environment, and the results of `body` are returned as the results of the procedure call.

It is an error for the arguments not to agree with the `formals` of any clause^c.

Example:

```
(define add1
  (case-lambda
    ((a) (add1 a 0))
    ((a b) (+ 1 a b))))
```



```
(add1 1) ;; => 2
(add1 1 2) ;; => 4
```



```
(scheme char)
```

```
(char-alphabetic? char)
```

TODO

```
(char-alphabetic? char)
```

TODO

```
(char-ci<=? char)
```

TODO

```
(char-ci<? char)
```

TODO

```
(char-ci=? char)
```

TODO

```
(char-ci>=? char)
```

TODO

```
(char-ci>? char)
```

TODO

(char-downcase char)

TODO

(char-foldcase char)

TODO

(char-lower-case? char)

TODO

(char-numeric? char)

TODO

(char-upcase char)

TODO

(char-upper-case? char)

TODO

(char-whitespace? char)

TODO

(string-ci<=? string1 string2 ...)

TODO

(string-ci<? string1 string2 ...)

TODO

(string-ci=? string1 string2 ...)

TODO

(string-ci>=? string1 string2 ...)

TODO

(string-ci>? string1 string2 ...)

TODO

(string-downcase string)

TODO

(string-foldcase string)

TODO

(string-upcase string)

TODO

(scheme complex)

angle

TODO

imag-part

TODO

magnitude

TODO

make-polar

TODO

```
make-rectangular
```

TODO

```
real-part
```

TODO

```
(scheme cxr)
```

Exports the following procedure which are the compositions of from three to four `car` and `cdr` operations. For example `caddar` could be defined by:

```
(define caddar
  (lambda (x) (car (cdr (cdr (car x))))))
```

Here is the full list:

- caaaar
- caaadrr
- caaar
- caadar
- caaddr
- caadr
- cadaar
- cadadr
- cedar
- caddar
- cadddr
- caddr
- cdaaar
- cdaadr
- cdaar
- cdadar
- cdaddr
- cdadr
- cdcaaar
- cddadr
- cddar
- cdddar
- cdddr

```
(scheme eval)

(environment list1 ...)
```

This procedure returns a specifier for the environment that results by starting with an empty environment and then importing each list, considered as an import set, into it. (See section 5.6 for a description of import sets.) The bindings of the environment represented by the specifier are immutable, as is the environment itself.

```
(eval expr-or-def environment-specifier)
```

If expr-or-def is an expression, it is evaluated in the specified environment and its values are returned. If it is a definition, the specified identifier(s) are defined in the specified environment, provided the environment is not immutable. Implementations may extend eval to allow other objects.

```
(scheme file)
```

```
(call-with-input-file)
```

TODO

```
(call-with-output-file)
```

TODO

```
(delete-file)
```

TODO

```
(file-exists?)
```

TODO

```
(open-input-file)
```

TODO

(open-output-file)

TODO

(with-input-from-file)

TODO

(with-output-to-file)

TODO

(open-binary-input-file)

TODO

(open-binary-output-file)

TODO

(scheme inexact)

(acos z)

TODO

(asin z)

TODO

(atan z)

TODO

(cos z)

TODO

(exp z t)

TODO

(finite? z)

TODO

(infinite? z)

TODO

(log z)

TODO

(nan? z)

TODO

sin

TODO

sqrt

TODO

tan

TODO

(scheme lazy)

(delay exp)

TODO

`(force promise)`

TODO

`(delay-force exp)`

TODO

`(promise? obj)`

TODO

`(make-promise exp)`

TODO

`(scheme load)`

`(load filename [environment])`

It is an error if filename is not a string.

An implementation-dependent operation is used to transform filename into the name of an existing file containing Scheme source code. The load procedure reads expressions and definitions from the file and evaluates them sequentially in the environment specified by environment-specifier. If environment-specifier is omitted, (interaction-environment) is assumed.

It is unspecified whether the results of the expressions are printed. The load procedure does not affect the values returned by current-input-port and current-output-port. It returns an unspecified value.

`(scheme process-context)`

`(command-line)`

Returns the command line passed to the process as a list of strings. The first string corresponds to the command name, and is implementation-dependent. It is an error to mutate any of these strings.

(emergency-exit [obj])

Terminates the program without running any outstanding dynamic-wind after procedures and communicates an exit value to the operating system in the same manner as exit.

(exit [obj])

Runs all outstanding dynamic-wind after procedures, terminates the running program, and communicates an exit value to the operating system. If no argument is supplied, or if obj is #t, the exit procedure should communicate to the operating system that the program exited normally. If obj is #f, the exit procedure should communicate to the operating system that the program exited abnormally. Otherwise, exit should translate obj into an appropriate exit value for the operating system, if possible.

(get-environment-variable name)

Many operating systems provide each running process with an environment consisting of environment variables. Both the name and value of an environment variable are strings. The procedure get-environment-variable returns the value of the environment variable name, or #f if the named environment variable is not found. It may use locale information to encode the name and decode the value of the environment variable. It is an error if get-environment-variable can't decode the value. It is also an error to mutate the resulting string.

(get-environment-variables)

Returns the names and values of all the environment variables as an alist, where the car of each entry is the name of an environment variable and the cdr is its value, both as strings. The order of the list is unspecified. It is an error to mutate any of these strings or the alist itself.

(scheme r5rs)

The library export R5RS forms. It is based on the following libraries:

- **(scheme base)**
- **(scheme inexact)**
- **(scheme complex)**
- **(scheme cxr)**
- **(scheme file)**
- **(scheme char)**

- (scheme read)
- (scheme write)
- (scheme eval)
- (scheme repl)
- (scheme load)
- (scheme lazy)t

It exports the following forms:

- *
- +
- -
- /
- <
- <=
- =
- >
- >=
- abs
- acos
- and
- angle
- append
- apply
- asin
- assoc
- assq
- assv
- atan
- begin
- boolean?
- caaaar
- caaadrr
- caaar
- caadar
- caaddr
- caadr
- caar
- cadaar
- cadadr
- cedar
- caddar
- cadddr
- caddr
- cadr
- call-with-current-continuation
- call-with-input-file

- call-with-output-file
- call-with-values
- car
- case
- cdaaar
- cdaadr
- cdaar
- cdadar
- cdaddr
- cdadr
- cdar
- cdhaar
- cddadr
- cddar
- cdddar
- cddddr
- cdddr
- cddr
- cdr
- ceiling
- char->integer
- char-alphabetic?
- char-ci<=?
- char-ci<?
- char-ci=?
- char-ci>=?
- char-ci>?
- char-downcase
- char-lower-case?
- char-numeric?
- char-ready?
- char-upcase
- char-upper-case?
- char-whitespace?
- char<=?
- char<?
- char=?
- char>=?
- char>?
- char?
- close-input-port
- close-output-port
- complex?
- cond
- cons
- cos

- current-input-port
- current-output-port
- define
- define-syntax
- delay
- denominator
- display
- do
- dynamic-wind
- eof-object?
- eq?
- equal?
- eqv?
- eval
- even?
- exact->inexact
- exact?
- exp
- expt
- floor
- for-each
- force
- gcd
- if
- imag-part
- inexact->exact
- inexact?
- input-port?
- integer->char
- integer?
- interaction-environment
- lambda
- lcm
- length
- let
- let*
- let-syntax
- letrec
- letrec-syntax
- list
- list->string
- list->vector
- list-ref
- list-tail
- list?
- load

- `log`
- `magnitude`
- `make-polar`
- `make-rectangular`
- `make-string`
- `make-vector`
- `map`
- `max`
- `member`
- `memq`
- `memv`
- `min`
- `modulo`
- `negative?`
- `newline`
- `not`
- `null-environment`
- `null?`
- `number->string`
- `number?`
- `numerator`
- `odd?`
- `open-input-file`
- `open-output-file`
- `or`
- `output-port?`
- `pair?`
- `peek-char`
- `positive?`
- `procedure?`
- `quasiquote`
- `quote`
- `quotient`
- `rational?`
- `rationalize`
- `read`
- `read-char`
- `real-part`
- `real?`
- `remainder`
- `reverse`
- `round`
- `scheme-report-environment`
- `set!`
- `set-car!`
- `set-cdr!`

- sin
- sqrt
- string
- string->list
- string->number
- string->symbol
- string-append
- string-ci<=?
- string-ci<?
- string-ci=?
- string-ci>=?
- string-ci>?
- string-copy
- string-fill!
- string-length
- string-ref
- string-set!
- string<=?
- string<?
- string=?
- string>=?
- string>?
- string?
- substring
- symbol->string
- symbol?
- syntax-rules
- tan
- truncate
- values
- vector
- vector->list
- vector-fill!
- vector-length
- vector-ref
- vector-set!
- vector?
- with-input-from-file
- with-output-to-file
- write
- write-char
- zero?

(scheme read)

(read [port])

The read procedure converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the non-terminal datum. It returns the next object parsable from the given textual input port, updating port to point to the first character past the end of the external representation of the object.

The current implementation is not fully compatible with R7RS.

(scheme repl)

(interaction-environment)

This procedure returns a specifier for a mutable environment that contains an implementation-defined set of bindings, typically a superset of those exported by (scheme base). The intent is that this procedure will return the environment in which the implementation would evaluate expressions entered by the user into a REPL.

(scheme time)

(current-jiffy)

Returns the number of jiffies as an exact integer that have elapsed since an arbitrary, implementation-defined epoch. A jiffy is an implementation-defined fraction of a second which is defined by the return value of the jiffies-per-second procedure. The starting epoch is guaranteed to be constant during a run of the program, but may vary between runs.

(current-second)

Returns an inexact number representing the current time on the International Atomic Time (TAI) scale. The value 0.0 represents midnight on January 1, 1970 TAI (equivalent to ten seconds before midnight Universal Time) and the value 1.0 represents one TAI second later. Neither high accuracy nor high precision are required; in particular, returning Coordinated Universal Time plus a suitable constant might be the best an implementation can do.

(jiffies-per-second)

Returns an exact integer representing the number of jiffies per SI second. This value is an implementation-specified constant.

(scheme write)

(display obj [port])

Writes a representation of obj to the given textual output port. Strings that appear in the written representation are output as if by write-string instead of by write. Symbols are not escaped. Character objects appear in the representation as if written by write-char instead of by write.

(write obj [port])

Writes a representation of obj to the given textual output port. Strings that appear in the written representation are enclosed in quotation marks, and within those strings backslash and quotation mark characters are escaped by backslashes. Symbols that contain non-ASCII characters are escaped with vertical lines. Character objects are written using the # notation.

If obj contains cycles which would cause an infinite loop using the normal written representation, then at least the objects that form part of the cycle must be represented using datum labels as described in section 2.4. Datum labels must not be used if there are no cycles.

(write-simple obj [port])

The write-simple procedure is the same as write, except that shared structure is never represented using datum labels. This can cause write-simple not to terminate if obj contains circular structure.

(write-shared obj [port])

The write-shared procedure is the same as write, except that shared structure must be represented using datum labels for all pairs and vectors that appear more than once in the output.

(scheme generator)

This is based on SRFI-158

Abstract

This SRFI defines utility procedures that create, transform, and consume generators. A generator is simply a procedure with no arguments that works as a source of values. Every time it is called, it yields a value. Generators may be finite or infinite; a finite generator returns an end-of-file object to indicate that it is exhausted. For example, `read-char`, `read-line`, and `read` are generators that generate characters, lines, and objects from the current input port. Generators provide lightweight laziness.

This SRFI also defines procedures that return accumulators. An accumulator is the inverse of a generator: it is a procedure of one argument that works as a sink of values.

Reference

Generator

(generator arg ...)

The simplest finite generator. Generates each of its arguments in turn. When no arguments are provided, it returns an empty generator that generates no values.

(circular-generator arg ...)

The simplest infinite generator. Generates each of its arguments in turn, then generates them again in turn, and so on forever.

(make-iota-generator count [start [step]])

Creates a finite generator of a sequence of count numbers. The sequence begins with start (which defaults to 0) and increases by step (which defaults to 1). If both start and step are exact, it generates exact numbers; otherwise it generates inexact numbers. The exactness of count doesn't affect the exactness of the results.

(make-range-generator start [end [step]])

Creates a generator of a sequence of numbers. The sequence begins with start, increases by step (default 1), and continues while the number is less than end, or forever if end is omitted. If both start and step are exact, it generates exact numbers; otherwise it generates inexact numbers. The exactness of end doesn't affect the exactness of the results.

```
(make-coroutine-generator proc)
```

Creates a generator from a coroutine.

The proc argument is a procedure that takes one argument, yield. When called, make-coroutine-generator immediately returns a generator g. When g is called, proc runs until it calls yield. Calling yield causes the execution of proc to be suspended, and g returns the value passed to yield.

Whether this generator is finite or infinite depends on the behavior of proc. If proc returns, it is the end of the sequence — g returns an end-of-file object from then on. The return value of proc is ignored.

The following code creates a generator that produces a series 0, 1, and 2 (effectively the same as (make-range-generator 0 3)) and binds it to g.

```
(define g
  (make-coroutine-generator
    (lambda (yield) (let loop ((i 0))
                      (when (< i 3) (yield i) (loop (+ i 1)))))))
```

```
(generator->list g) ;;=> (0 1 2)
```

```
(list->generator list)
```

Convert LIST into a generator.

```
(vector->generator vector [start [end]])
```

```
(reverse-vector->generator vector [start [end]])
```

```
(string->generator string [start [end]])
```

```
(bytevector->generator bytevector [start [end]])
```

These procedures return generators that yield each element of the given argument. Mutating the underlying object will affect the results of the generator.

```
(generator->list (list->generator '(1 2 3 4 5)))
 ;;=> (1 2 3 4 5)
(generator->list (vector->generator '#(1 2 3 4 5)))
 ;;=> (1 2 3 4 5)
(generator->list (reverse-vector->generator '#(1 2 3 4 5)))
 ;;=> (5 4 3 2 1)
(generator->list (string->generator "abcde"))
 ;;=> (#\a #\b #\c #\d #\e)
```

The generators returned by the constructors are exhausted once all elements are retrieved; the optional start-th and end-th arguments can limit the range the generator walks across.

For reverse-vector->generator, the first value is the element right before the end-th element, and the last value is the start-th element. For all the other constructors, the first value the generator yields is the start-th element, and it ends right before the end-th element.

```
(generator->list (vector->generator '#(a b c d e) 2))
;; => (c d e)
(generator->list (vector->generator '#(a b c d e) 2 4))
;; => (c d)
(generator->list (reverse-vector->generator '#(a b c d e) 2))
;; => (e d c)
(generator->list (reverse-vector->generator '#(a b c d e) 2 4))
;; => (d c)
(generator->list (reverse-vector->generator '#(a b c d e) 0 2))
;; => (b a)

(make-for-each-generator for-each obj)
```

A generator constructor that converts any collection obj to a generator that returns its elements using a for-each procedure appropriate for obj. This must be a procedure that when called as (for-each proc obj) calls proc on each element of obj. Examples of such procedures are for-each, string-for-each, and vector-for-each from R7RS. The value returned by for-each is ignored. The generator is finite if the collection is finite, which would typically be the case.

The collections need not be conventional ones (lists, strings, etc.) as long as for-each can invoke a procedure on everything that counts as a member. For example, the following procedure allows for-each-generator to generate the digits of an integer from least to most significant:

```
(define (for-each-digit proc n)
  (when (> n 0)
    (let-values (((div rem) (truncate/ n 10)))
      (proc rem)
      (for-each-digit proc div))))
```



```
(make-unfold-generator stop? mapper successor seed)
```

A generator constructor similar to (scheme list) unfold.

The stop? predicate takes a seed value and determines whether to stop. The mapper procedure calculates a value to be returned by the generator from a seed value. The successor procedure calculates the next seed value from the current seed value.

For each call of the resulting generator, stop? is called with the current seed value. If it returns true, then the generator returns an end-of-file object. Otherwise, it applies mapper to the current seed value to get the value to return, and uses successor to update the seed value.

This generator is finite unless stop? never returns true.

```
(generator->list (make-unfold-generator
                      (lambda (s) (> s 5))
                      (lambda (s) (* s 2))
                      (lambda (s) (+ s 1))
                      0))
;; => (0 2 4 6 8 10)
```

(gcons* item ... generator)

Returns a generator that adds items in front of gen. Once the items have been consumed, the generator is guaranteed to tail-call gen.

```
(generator->list (gcons* 'a 'b (make-range-generator 0 2)))
;; => (a b 0 1)
```

(gappend generator ...)

Returns a generator that yields the items from the first given generator, and once it is exhausted, from the second generator, and so on.

```
(generator->list (gappend (make-range-generator 0 3) (make-range-generator 0 2)))
;; => (0 1 2 0 1)
```

```
(generator->list (gappend))
;; => ()
```

(gflatten generator)

Returns a generator that yields the elements of the lists produced by the given generator.

```
(ggroup generator k [padding])
```

Returns a generator that yields lists of k items from the given generator. If fewer than k elements are available for the last list, and padding is absent, the short list is returned; otherwise, it is padded by padding to length k.

```
(gmerge less-than generator1 ...)
```

Returns a generator that yields the items from the given generators in the order dictated by less-than. If the items are equal, the leftmost item is used first. When all of given generators are exhausted, the returned generator is exhausted also.

As a special case, if only one generator is given, it is returned.

```
(gmap proc generator ...)
```

When only one generator is given, returns a generator that yields the items from the given generator after invoking proc on them.

When more than one generator is given, each item of the resulting generator is a result of applying proc to the items from each generator. If any of input generator is exhausted, the resulting generator is also exhausted.

Note: This differs from generator-map->list, which consumes all values at once and returns the results as a list, while gmap returns a generator immediately without consuming input.

```
(generator->list (gmap - (make-range-generator 0 3)))
;; => (0 . 1 . 2)

(generator->list (gmap cons (generator 1 2 3) (generator 4 5)))
;; => ((1 . 4) (2 . 5))
```

```
(gcombine proc seed generator generator2)
```

A generator for mapping with state. It yields a sequence of sub-folds over proc.

The proc argument is a procedure that takes as many arguments as the input generators plus one. It is called as (proc v1 v2 ... seed), where v1, v2, ... are the values yielded from the input generators, and seed is the current seed value. It must return two values, the yielding value and the next seed. The result generator is exhausted when any of the genn generators is exhausted, at which time all the others are in an undefined state.

```
(gfilter predicate generator)
```

```
(gremove predicate generator)
```

Returns generators that yield the items from the source generator, except those on which pred answers false or true respectively.

```
(gstate-filter proc seed generator)
```

Returns a generator that obtains items from the source generator and passes an item and a state (whose initial value is seed) as arguments to proc. Proc in turn returns two values, a boolean and a new value of the state. If the boolean is true, the item is returned; otherwise, this algorithm is repeated until gen is exhausted, at which point the returned generator is also exhausted. The final value of the state is discarded.

```
(gtake gen k [padding])
```

```
(gdrop gen k)
```

These are generator analogues of SRFI 1 take and drop. Gtake returns a generator that yields (at most) the first k items of the source generator, while gdrop returns a generator that skips the first k items of the source generator.

These won't complain if the source generator is exhausted before generating k items. By default, the generator returned by gtake terminates when the source generator does, but if you provide the padding argument, then the returned generator will yield exactly k items, using the padding value as needed to provide sufficient additional values.

```
gtake-while pred gen
```

```
gdrop-while pred gen
```

The generator analogues of SRFI-1 take-while and drop-while. The generator returned from gtake-while yields items from the source generator as long as pred returns true for each. The generator returned from gdrop-while first reads and discards values from the source generator while pred returns true for them, then starts yielding items returned by the source.

```
(gdelete item gen [=])
```

Creates a generator that returns whatever gen returns, except for any items that are the same as item in the sense of =, which defaults to equal?. The = predicate is passed exactly two arguments, of which the first was generated by gen before the second.

```
(generator->list (gdelete 3 (generator 1 2 3 4 5 3 6 7)))  
;; => (1 2 4 5 6 7)
```

```
(gdelete-neighbor-dups gen [=])
```

Creates a generator that returns whatever gen returns, except for any items that are equal to the preceding item in the sense of `=`, which defaults to `equal?.` The `=` predicate is passed exactly two arguments, of which the first was generated by gen before the second.

```
(generator->list (gdelete-neighbor-dups (list->generator '(a a b c a a a d c))))  
;; => (a b c a d c)
```

```
(gindex value-gen index-gen)
```

Creates a generator that returns elements of value-gen specified by the indices (non-negative exact integers) generated by index-gen. It is an error if the indices are not strictly increasing, or if any index exceeds the number of elements generated by value-gen. The result generator is exhausted when either generator is exhausted, at which time the other is in an undefined state.

```
(generator->list (gindex (list->generator '(a b c d e f))  
                           (list->generator '(0 2 4))))  
;; => (a c e)
```

```
(gselect value-gen truth-gen)
```

Creates a generator that returns elements of value-gen that correspond to the values generated by truth-gen. If the current value of truth-gen is true, the current value of value-gen is generated, but otherwise not. The result generator is exhausted when either generator is exhausted, at which time the other is in an undefined state.

```
(generator->list (gselect (list->generator '(a b c d e f))  
                           (list->generator '#t #f #f #t #t #f)))  
;; => (a d e)
```

```
(generator->list generator [k])
```

Reads items from generator and returns a newly allocated list of them. By default, it reads until the generator is exhausted.

If an optional argument k is given, it must be a non-negative integer, and the list ends when either k items are consumed, or generator is exhausted; therefore generator can be infinite in this case.

```
(generator->reverse-list generator [k])
```

Reads items from generator and returns a newly allocated list of them in reverse order. By default, this reads until the generator is exhausted.

If an optional argument k is given, it must be a non-negative integer, and the list ends when either k items are read, or generator is exhausted; therefore generator can be infinite in this case.

```
(generator->vector generator [k])
```

Reads items from generator and returns a newly allocated vector of them. By default, it reads until the generator is exhausted.

If an optional argument k is given, it must be a non-negative integer, and the list ends when either k items are consumed, or generator is exhausted; therefore generator can be infinite in this case.

```
(generator->vector! vector at generator)
```

Reads items from generator and puts them into vector starting at index at, until vector is full or generator is exhausted. Generator can be infinite. The number of elements generated is returned.

```
(generator->string generator [k])
```

Reads items from generator and returns a newly allocated string of them. It is an error if the items are not characters. By default, it reads until the generator is exhausted.

If an optional argument k is given, it must be a non-negative integer, and the string ends when either k items are consumed, or generator is exhausted; therefore generator can be infinite in this case.

```
(generator-fold proc seed generator ...)
```

Works like (scheme list) fold on the values generated by the generator arguments.

When one generator is given, for each value v generated by gen, proc is called as (proc v r), where r is the current accumulated result; the initial value of the accumulated result is seed, and the return value from proc becomes the next accumulated result. When gen is exhausted, the accumulated result at that time is returned from generator-fold.

When more than one generator is given, proc is invoked on the values returned by all the generator arguments followed by the current accumulated result. The procedure terminates when any of the generators is exhausted, at which time all the others are in an undefined state.

```
(with-input-from-string "a b c d e"
  (lambda () (generator-fold cons 'z read)))
  ;;= > (e d c b a . z)
```

(generator-for-each proc generator ...)

A generator analogue of for-each that consumes generated values using side effects. Repeatedly applies proc on the values yielded by gen, gen2 ... until any one of the generators is exhausted, at which time all the others are in an undefined state. The values returned from proc are discarded. Returns an unspecified value.

(generator-map->list proc generator ...)

A generator analogue of map that consumes generated values, processes them through a mapping function, and returns a list of the mapped values. Repeatedly applies proc on the values yielded by gen, gen2 ... until any one of the generators is exhausted, at which time all the others are in an undefined state. The values returned from proc are accumulated into a list, which is returned.

(generator-find predicate generator)

Returns the first item from the generator gen that satisfies the predicate pred, or #f if no such item is found before gen is exhausted. If gen is infinite, generator-find will not return if it cannot find an appropriate item.

(generator-count predicate generator)

Returns the number of items available from the generator gen that satisfy the predicate pred.

(generator-any predicate generator)

Applies predicate to each item from gen. As soon as it yields a true value, the value is returned without consuming the rest of gen. If gen is exhausted, returns #f.

(generator-every predicate generator)

Applies pred to each item from gen. As soon as it yields a false value, the value is returned without consuming the rest of gen. If gen is exhausted, returns the last value returned by pred, or #t if pred was never called.

(generator-unfold gen unfold arg ...)

Equivalent to (unfold eof-object? (lambda (x) x) (lambda (x) (gen)) (gen) arg ...). The values of gen are unfolded into the collection that unfold creates.

The signature of the unfold procedure is (unfold stop? mapper successor seed args ...). Note that the vector-unfold and vector-unfold-right of SRFI 43 and SRFI 133 do not have this signature and cannot be used with this procedure. To unfold into a vector, use SRFI 1's unfold and then apply list->vector to the result.

```
; Iterates over string and unfolds into a list using SRFI 1 unfold
(generator-unfold (make-for-each-generator string-for-each "abc") unfold)
; => (#\a #\b #\c)
```

Accumulator

(make-accumulator kons knil finalizer)

Returns an accumulator that, when invoked on an object other than an end-of-file object, invokes kons on its argument and the accumulator's current state, using the same order as a function passed to fold. It then sets the accumulator's state to the value returned by kons and returns an unspecified value. The initial state of the accumulator is set to knil. However, if an end-of-file object is passed to the accumulator, it returns the result of tail-calling the procedure finalizer on the state. Repeated calls with an end-of-file object will reinvoke finalizer.

(count-accumulator)

Returns an accumulator that, when invoked on an object, adds 1 to a count inside the accumulator and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the count.

(list-accumulator)

Returns an accumulator that, when invoked on an object, adds that object to a list inside the accumulator in order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the list.

(reverse-list-accumulator)

Returns an accumulator that, when invoked on an object, adds that object to a list inside the accumulator in reverse order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the list.

(vector-accumulator)

Returns an accumulator that, when invoked on an object, adds that object to a vector inside the accumulator in order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the vector.

(reverse-vector-accumulator)

Returns an accumulator that, when invoked on an object, adds that object to a vector inside the accumulator in reverse order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the vector.

(vector-accumulator! vector at)

Returns an accumulator that, when invoked on an object, adds that object to consecutive positions of vector starting at at in order of accumulation. It is an error to try to accumulate more objects than vector will hold. An unspecified value is returned. However, if an end-of-file object is passed, the accumulator returns vector.

(string-accumulator)

Returns an accumulator that, when invoked on a character, adds that character to a string inside the accumulator in order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the string.

(bytewise-accumulator)

Returns an accumulator that, when invoked on a byte, adds that integer to a bytewise inside the accumulator in order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the bytewise.

(bytewise-accumulator! bytewise at)

Returns an accumulator that, when invoked on a byte, adds that byte to consecutive positions of bytewise starting at at in order of accumulation. It is an error to try to accumulate more bytes than vector will hold. An unspecified value is returned. However, if an end-of-file object is passed, the accumulator returns bytewise.

(sum-accumulator)

Returns an accumulator that, when invoked on a number, adds that number to a sum inside the accumulator in order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the sum.

'(product-accumulator)

Returns an accumulator that, when invoked on a number, multiplies that number to a product inside the accumulator in order of accumulation and returns an unspecified value. However, if an end-of-file object is passed, the accumulator returns the product.