nomunofu

status: wip

Querying wikidata made easy

two origami frogs, one on top of the other

nomunofu is database server written in Scheme programming language that is powered by WiredTiger ordered key-value store, based on SRFI-167 and SRFI-168.

It allows to store and query triples, quads **and more**. The goal is to make it much easier, definitely faster to query as much as possible tuples. To achieve that goal, the server part of the database is made very simple, and it only knows how to do pattern matching and count, sum and average aggregation. Also, it is possible to swap the storage engine to something that is horizontally scalable and resilient (read: FoundationDB).

The *thin server*, *thick client* paradigm was choosen to allow the end-user to more easily workaround bugs in the data, and it also allows to offload the servers hosting the data from heavy computations.

Happy hacking!

Amirouche ~ zig ~ https://hyper.dev

`(scheme base)`

`_`

TODO (missing in r7rs?)

`...`

It is called ellipsis. It signify that a pattern must be repeated.

`=>`

TODO

`else`

Used in `cond` and `case` 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 ...) syntax`

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 is 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 return `#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 arethe 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 bytevector index)`

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

`(bytevector-u8-set! bytevector index byte)`

Stores `BYTE` as the `K`th 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 obj)`

TODO

`(cadr obj)`

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 proc)`

Abbreviation for `call-with-continuation`.

`(car obj)`

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

`(case <key> <clause1> <clause2> ...)` syntax

TODO

`(cdar obj)`

TODO

`(cddr obj)`

TODO

`(cdr obj)`

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<=? char1 char2)`

TODO

`(char<? char1 char2)`

TODO

`(char=? char1 char2)`

TODO

`(char>=? char1 char2)`

TODO

`(char>? char1 char2)`

TODO

(char? obj)

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? obj)

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

(cond <clause1> ...)

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 [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 [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 <name> <expr>)` syntax

TODO

`(define (<name> <variable> ...) <expr> ...)` syntax

TODO

`define-record-type` syntax

TODO

`define-syntax` 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

`(file-error? error)`

TODO

`(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 port)`

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 <clause> ...) syntax`

TODO

`(if <expr> <then> [<else>])`

TODO

`include`

TODO

`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 <formals> <expr> ...)`

TODO

`(lcm n1 ...)`

Return the least common multiple of its arguments.

`(length list)`

Returns the length of list.

`let` syntax

TODO

`let*` syntax

TODO

`let*-values` syntax

TODO

`let-syntax` syntax

TODO

`let-values` syntax

TODO

`letrec` syntax

TODO

`letrec*` syntax

TODO

`letrec-syntax` 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])

he 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` syntax

TODO

`quote` syntax

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 straing [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? obj)

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` syntax

TODO

`unquote-splicing` syntax

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? obj)`

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> ...) 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‘.

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:

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

Here is the full list:

- `caaaar`
- `caaadr`
- `caaar`
- `caadar`
- `caaddr`
- `caadr`
- `cadaar`
- `cadadr`
- `cadar`
- `caddar`
- `cadddr`
- `caddr`
- `cdaaar`
- `cdaadr`
- `cdaar`
- `cdadar`
- `cdaddr`
- `cdadr`
- `cddaar`
- `cddadr`
- `cddar`
- `cdddar`
- `cddddr`
- `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. 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 z)

TODO

`(sqrt z)`

TODO

`(tan z)`

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`. If `ENVIRONMENT` 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)`

Thie 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`
- `caaadr`
- `caaar`
- `caadar`
- `caaddr`
- `caadr`
- `caar`
- `cadaar`
- `cadadr`
- `cadar`
- `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`
- `cddaar`
- `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 bitwise)`

Re-export SRFI-151.

`(scheme box)`

Re-export SRFI-111.

`(scheme bytevector)`

This is based on R6RS bytevectors

Abstract

TODO

Reference

General operations

`(endianness <endianess symbol>) syntax`

`(native-endianness)`

Returns the endianness symbol associated implementation's preferred endianness (usually that of the underlying machine architecture). This may be any , including a symbol other than big and little.

`(bytevector? obj)`

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

`(make-bytevector k [fill])`

Returns a newly allocated bytevector of `K` bytes.

If the `FILL` argument is missing, the initial contents of the returned bytevector are unspecified.

If the `FILL` argument is present, it must be an exact integer object in the interval {-128, … 255} that specifies the initial value for the bytes of the bytevector: If `FILL` is positive, it is interpreted as an octet; if it is negative, it is interpreted as a byte.

`(bytevector-length bytevector)`

Returns, as an exact integer object, the number of bytes in bytevector.

`(bytevector=? bytevector1 bytevector2)`

Returns #t if bytevector1 and bytevector2 are equal-that is, if they have the same length and equal bytes at all valid indices. It returns #f otherwise.

`(bytevector-fill! bytevector fill)`

The fill argument is as in the description of the make-bytevector procedure. The bytevector-fill! procedure stores fill in every element of bytevector and returns unspecified values. Analogous to vector-fill!.

`(bytevector-copy! source source-start target target-start k)`

`(bytevector-copy bytevector)`

Returns a newly allocated copy of bytevector.

Operations on bytes and octets

`(bytevector-u8-ref bytevector k)`

The bytevector-u8-ref procedure returns the byte at index k of bytevector, as an octet.

`(bytevector-s8-ref bytevector k)`

The bytevector-s8-ref procedure returns the byte at index k of bytevector, as a (signed) byte.

`(bytevector-u8-set! bytevector k octet)`

The bytevector-u8-set! procedure stores octet in element k of bytevector.

`(bytevector-s8-set! bytevector k byte)`

The bytevector-s8-set! procedure stores the two's-complement representation of byte in element k of bytevector.

`(bytevector->u8-list bytevector)`

The bytevector->u8-list procedure returns a newly allocated list of the octets of bytevector in the same order.

`(u8-list->bytevector list)`

The u8-list->bytevector procedure returns a newly allocated bytevector whose elements are the elements of list list, in the same order. It is analogous to list->vector.

Operations on integers of arbitrary size

```
(bytevector-uint-ref bytevector k endianness size)
```

```
(bytevector-sint-ref bytevector k endianness size)
```

```
(bytevector-uint-set! bytevector k n endianness size)
```

```
(bytevector-sint-set! bytevector k n endianness size)
```

```
(bytevector->uint-list bytevector endianness size)
```

```
(bytevector->sint-list bytevector endianness sizee
```

```
(uint-list->bytevector list endianness size)
```

```
(sint-list->bytevector list endianness size)
```

Operations on 16-bit integers

```
(bytevector-u16-ref bytevector k endianness)
```

```
(bytevector-s16-ref bytevector k endianness)
```

```
(bytevector-u16-native-ref bytevector k)
```

```
(bytevector-s16-native-ref bytevector k)
```

```
(bytevector-u16-set! bytevector k n endianness)
```

```
(bytevector-s16-set! bytevector k n endianness)
```

```
(bytevector-u16-native-set! bytevector k n)
```

```
(bytevector-s16-native-set! bytevector k n)
```

Operations on 32-bit integers

`(bytevector-u32-ref bytevector k endianness)`

`(bytevector-s32-ref bytevector k endianness)`

`(bytevector-u32-native-ref bytevector k)`

`(bytevector-s32-native-ref bytevector k)`

`(bytevector-u32-set! bytevector k n endianness)`

`(bytevector-s32-set! bytevector k n endianness)`

`(bytevector-u32-native-set! bytevector k n)`

`(bytevector-s32-native-set! bytevector k n)`

Operations on 64-bit integers

`(bytevector-u64-ref bytevector k endianness)`

`(bytevector-s64-ref bytevector k endianness)`

`(bytevector-u64-native-ref bytevector k)`

`(bytevector-s64-native-ref bytevector k)`

`(bytevector-u64-set! bytevector k n endianness)`

`(bytevector-s64-set! bytevector k n endianness)`

`(bytevector-u64-native-set! bytevector k n)`

`(bytevector-s64-native-set! bytevector k n)`

Operations on IEEE-754 representations

`(bytevector-ieee-single-native-ref bytevector k)`

`(bytevector-ieee-single-ref bytevector k endianness)`

`(bytevector-ieee-double-native-ref bytevector k)`

`(bytevector-ieee-double-ref bytevector k endianness)`

`(bytevector-ieee-single-native-set! bytevector k x)`

`(bytevector-ieee-single-set! bytevector k x endianness)`

`(bytevector-ieee-double-native-set! bytevector k x)`

`(bytevector-ieee-double-set! bytevector k x endianness)`

Operations on strings

`(string->utf8 string)`

`(string->utf16 string)`

`(string->utf16 string endianness)`

`(string->utf32 string)`

`(string->utf32 string endianness)`

`(utf8->string bytevector)`

`(utf16->string bytevector endianness)`

`(utf16->string bytevector endianness endianness-mandatory)`

`(utf32->string bytevector endianness)`

`(utf32->string bytevector endianness endianness-mandatory)`

`(scheme charset)`

Re-export SRFI-14.

`(scheme comparator)`

Re-export SRFI-128.

`(scheme division)`

Re-export SRFI-141.

`(scheme ephemeron)`

Re-export SRFI-124.

`(scheme generator)`

Re-export SRFI-158.

`(scheme hash-table)`

Re-export SRFI-125.

`(scheme idque)`

Re-export SRFI-134.

`(scheme ilist)`

Re-export SRFI-116.

`(scheme list-queue)`

Re-export SRFI-117.

`(scheme list)`

Re-export SRFI-1.

`(scheme lseq)`

Re-export SRFI-127.

`(scheme mapping)`

Re-export SRFI-146.

`(scheme mapping hash)`

Re-export SRFI-146.

`(scheme regex)`

Re-export SRFI-115.

`(scheme rlist)`

Re-export SRFI-101.

`(scheme set)`

Re-export SRFI-113.

`(scheme show)`

Re-export SRFI-159.

`(scheme sort)`

Re-export SRFI-132.

`(scheme stream)`

Re-export SRFI-41.

`(scheme text)`

Re-export SRFI-135.

`(scheme vector)`

Re-export SRFI-160.

`(scheme vector @)`

Re-export SRFI-160.

`(srfi srfi-1)`

This is based on SRFI-1.

Abstract

TODO

Reference

Constructors

`(cons a d)`

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

`(list object ...)`

Returns a newly allocated list of its arguments.

`(xcons d a)`

```
(lambda (d a) (cons a d))
```

Of utility only as a value to be conveniently passed to higher-order procedures.

```
(xcons '(b c) 'a) ;; => (a b c)
```

The name stands for “eXchanged CONS.”

`(cons* obj ... tail)`

Like list, but the last argument provides the tail of the constructed list.

`(make-list n [fill])`

Returns an n-element list, whose elements are all the value fill. If the fill argument is not given, the elements of the list may be arbitrary values.

```
(make-list 4 'c) => (c c c c)
```

`(list-tabulate n init-proc)`

Returns an n-element list. Element i of the list, where 0 <= i < n, is produced by (init-proc i). No guarantee is made about the dynamic order in which init-proc is applied to these indices.

```
(list-tabulate 4 values) => (0 1 2 3)
```

`(list-copy flist)`

Copies the spine of the argument.

`(circular-list elt1 elt2 ...)`

Constructs a circular list of the elements.

```
(circular-list 'z 'q) => (z q z q z q ...)
```

`(iota count [start step])`

Returns a list containing the elements:

```
(start start+step ... start+(count-1)*step)
```

The start and step parameters default to 0 and 1, respectively.

```
(iota 5) => (0 1 2 3 4)
(iota 5 0 -0.1) => (0 -0.1 -0.2 -0.3 -0.4)
```

Predicates

`(proper-list? x)`

Returns true iff x is a proper list – a finite, nil-terminated list.

More carefully: The empty list is a proper list. A pair whose cdr is a proper list is also a proper list. The opposite of proper is improper.

`(circular-list? x)`

True if x is a circular list. A circular list is a value such that for every n >= 0, cdrn(x) is a pair.

Terminology: The opposite of circular is finite.

'(dotted-list? x)

True if x is a finite, non-nil-terminated list. That is, there exists an n >= 0 such that cdrn(x) is neither a pair nor (). This includes non-pair, non-() values (e.g. symbols, numbers), which are considered to be dotted lists of length 0.

`(pair? obj)`

Returns #t if object is a pair; otherwise, #f.

`(null? obj)`

Returns #t if object is the empty list; otherwise, #f.

`(null-list? list)`

List is a proper or circular list. This procedure returns true if the argument is the empty list (), and false otherwise. It is an error to pass this procedure a value which is not a proper or circular list. This procedure is recommended as the termination condition for list-processing procedures that are not defined on dotted lists.

‘(not-pair? x)

Provided as a procedure as it can be useful as the termination condition for list-processing procedures that wish to handle all finite lists, both proper and dotted.

`(list= elt= list1 ...)`

Determines list equality, given an element-equality procedure.

Selectors

(car pair)

(cdr pair)

These functions return the contents of the car and cdr field of their argument, respectively. Note that it is an error to apply them to the empty list.

Also the following selectors are defined:

- `caar`
- `cadr`
- `cdar`
- `cddr`
- `caaar`
- `caadr`
- `cadar`
- `caddr`
- `cdaar`
- `cdadr`
- `cddar`
- `cdddr`
- `caaaar`
- `caaadr`
- `caadar`
- `caaddr`
- `cadaar`
- `cadadr`
- `caddar`
- `cadddr`
- `cdaaar`

- `cdaadr`
- `cdadar`
- `cdaddr`
- `cddaar`
- `cddadr`
- `cdddar`
- `cddddr`

`(list-ref clist i)`

Returns the ith element of clist. (This is the same as the car of (drop clist i).) It is an error if i >= n, where n is the length of clist.

```
(list-ref '(a b c d) 2) => c
```

`(first pair)`

`(second pair)`

`(third pair)`

`(fourth pair)`

`(fifth pair)`

`(sixth pair)`

`(seventh pair)`

`(eighth pair)`

`(ninth pair)`

`(tenth pair)`

Synonyms for `car`, `cadr`, `caddr`, ...

`(car+cdr pair)`

The fundamental pair deconstructor:

```
(lambda (p) (values (car p) (cdr p)))
```

This can, of course, be implemented more efficiently by a compiler.

`(take lst i)`

`(drop lst i)`

`take` returns the first `I` elements of list `LST`. `drop` returns all but the first i elements of list `LST`.

```
(take '(a b c d e) 2) ;; => (a b)
(drop '(a b c d e) 2) ;; => (c d e)
```

`LST` may be any value – a proper, circular, or dotted list:

```
(take '(1 2 3 . d) 2) ;; => (1 2)
(drop '(1 2 3 . d) 2) ;; => (3 . d)
(take '(1 2 3 . d) 3) ;; => (1 2 3)
(drop '(1 2 3 . d) 3) ;; => d
```

For a legal `I`, `take` and `drop` partition the list in a manner which can be inverted with append:

```
(equal? (append (take lst i) (drop x i)) lst)
```

`drop` is exactly equivalent to performing `i` `cdr` operations on `LST`; the returned value shares a common tail with `LST`. If the argument is a list of non-zero length, `take` is guaranteed to return a freshly-allocated list, even in the case where the entire list is taken, e.g. `(take lst (length lst))`.

`(take-right flist i)`

`(drop-right flist i)`

`take-right` returns the last `I` elements of `FLIST`. `drop-right` returns all but the last `I` elements of `FLIST`.

```
(take-right '(a b c d e) 2) => (d e)
(drop-right '(a b c d e) 2) => (a b c)
```

The returned list may share a common tail with the argument list.

`FLIST` may be any finite list, either proper or dotted:

```
(take-right '(1 2 3 . d) 2) => (2 3 . d)
(drop-right '(1 2 3 . d) 2) => (1)
(take-right '(1 2 3 . d) 0) => d
(drop-right '(1 2 3 . d) 0) => (1 2 3)
```

For a legal `I`, `take-right` and `drop-right` partition the list in a manner which can be inverted with `append`:

```
(equal? (append (take flist i) (drop flist i)) flist)
```

`take-right`'s return value is guaranteed to share a common tail with `FLIST`. If the argument is a list of non-zero length, `drop-right` is guaranteed to return a freshly-allocated list, even in the case where nothing is dropped, e.g. `(drop-right flist 0)`.

`(take! x i)`

`(drop-right! flist i)`

`take!` and `drop-right!` are "linear-update" variants of `take` and `drop-right`: the procedure is allowed, but not required, to alter the argument list to produce the result.

If `x` is circular, `take!` may return a shorter-than-expected list:

```
(take! (circular-list 1 3 5) 8) => (1 3)
(take! (circular-list 1 3 5) 8) => (1 3 5 1 3 5 1 3)
```

`(split-at  x i)`

`(split-at! x i)`

`split-at` splits the list `x` at index `i`, returning a list of the first `i` elements, and the remaining tail. It is equivalent to:

```
(values (take x i) (drop x i))
```

`split-at!` is the linear-update variant. It is allowed, but not required, to alter the argument list to produce the result.

```
(split-at '(a b c d e f g h) 3) ;; => (a b c) and (d e f g h)
```

`(last pair)`

`(last-pair pair)`

`last` returns the last element of the non-empty, finite list `pair`. `last-pair` returns the last pair in the non-empty, finite list `pair`:

```
(last '(a b c)) ;; => c
(last-pair '(a b c)) ;; => (c)
```

Miscellaneous

`(length list)`

`(length+ clist)`

Both `length` and `length+` return the length of the argument. It is an error to pass a value to length which is not a proper list (finite and nil-terminated). In particular, this means an implementation may diverge or signal an error when length is applied to a circular list.

`length+`, on the other hand, returns `#f` when applied to a circular list.

The length of a proper list is a non-negative integer `n` such that `cdr` applied `n` times to the list produces the empty list.

`(append  list1 ...)`

`(append! list1 ...)`

`append` returns a list consisting of the elements of `list1` followed by the elements of the other list parameters.

```
(append '(x) '(y))        =>  (x y)
(append '(a) '(b c d))    =>  (a b c d)
(append '(a (b)) '((c)))  =>  (a (b) (c))
```

The resulting list is always newly allocated, except that it shares structure with the final list argument. This last argument may be any value at all; an improper list results if it is not a proper list. All other arguments must be proper lists.

```
(append '(a b) '(c . d))  =>  (a b c . d)
(append '() 'a)           =>  a
(append '(x y))           =>  (x y)
(append)                  =>  ()
```

`append!` is the “linear-update” variant of append – it is allowed, but not required, to alter `cons` cells in the argument lists to construct the result list. The last argument is never altered; the result list shares structure with this parameter.

`(concatenate  list-of-lists)`

`(concatenate! list-of-lists)`

These functions append the elements of their argument together. That is, `concatenate` returns:

```
(apply append list-of-lists)
```

Or, equivalently,

```
(reduce-right append '() list-of-lists)
```

`concatenate!` is the linear-update variant, defined in terms of `append!` instead of `append`.

As with `append` and `append!`, the last element of the input list may be any value at all.

`(reverse list)`

`(reverse! list)`

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

```
(reverse '(a b c)) ;; =>  (c b a)
(reverse '(a (b c) d (e (f)))) ;; =>  ((e (f)) d (b c) a)
```

`reverse!` is the linear-update variant of reverse. It is permitted, but not required, to alter the argument's `cons` cells to produce the reversed list.

`(append-reverse rev-head tail)`

`(append-reverse! rev-head tail)`

`append-reverse` returns `(append (reverse rev-head) tail)`. It is provided because it is a common operation – a common list-processing style calls for this exact operation to transfer values accumulated in reverse order onto the front of another list, and because the implementation is significantly more efficient than the simple composition it replaces. (But note that this pattern of iterative computation followed by a `reverse` can frequently be rewritten as a recursion, dispensing with the `reverse` and `append-reverse` steps, and shifting temporary, intermediate storage from the heap to the stack, which is typically a win for reasons of cache locality and eager storage reclamation.)

`append-reverse!` is just the linear-update variant – it is allowed, but not required, to alter rev-head's `cons` cells to construct the result.

(zip clist1 clist2 ...)

```
(lambda lists (apply map list lists))
```

If `zip` is passed n lists, it returns a list as long as the shortest of these lists, each element of which is an n-element list comprised of the corresponding elements from the parameter lists.

```
(zip '(one two three)
     '(1 2 3)
     '(odd even odd even odd even odd even))
     ;; => ((one 1 odd) (two 2 even) (three 3 odd))

(zip '(1 2 3)) => ((1) (2) (3))
```

(unzip1 list)

(unzip2 list)

(unzip3 list)

(unzip4 list)

(unzip5 list)

`unzip1` takes a list of lists, where every list must contain at least one element, and returns a list containing the initial element of each such list. That is, it returns `(map car lists)`. `unzip2` takes a list of lists, where every list must contain at least two elements, and returns two values: a list of the first elements, and a list of the second elements. `unzip3` does the same for the first three elements of the lists, and so forth.

```
(unzip2 '((1 one) (2 two) (3 three))) ;; => '((1 2 3) (one two three))
```

(count pred clist1 ...)

`pred` is a procedure taking as many arguments as there are lists and returning a single value. It is applied element-wise to the elements of the lists, and a count is tallied of the number of elements that produce a true value. This count is returned. `count` is "iterative" in that it is guaranteed to apply `pred` to the list elements in a left-to-right order. The counting stops when the shortest list expires.

```
(count even? '(3 1 4 1 5 9 2 5 6)) => 3
(count < '(1 2 4 8) '(2 4 6 8 10 12 14 16)) => 3
```

At least one of the argument lists must be finite:

```
(count < '(3 1 4 1) (circular-list 1 10)) => 2
```

Fold, unfold & map

`(fold kons knil list1 ...)`

TODO

`(fold-right kons knil list1 ...)`

TODO

`(pair-fold kons knil list1 ...)`

TODO

`(pair-fold-right kons knil list1 ...)`

TODO

`(reduce f ridentity list)`

`reduce` is a variant of `fold`.

`ridentity` should be a "right identity" of the procedure `f` – that is, for any value `x` acceptable to `f`:

```
(f x ridentity) ;; => x
```

Note: that `ridentity` is used only in the empty-list case. You typically use reduce when applying `f` is expensive and you'd like to avoid the extra application incurred when fold applies `f` to the head of list and the identity value, redundantly producing the same value passed in to `f`. For example, if `f` involves searching a file directory or performing a database query, this can be significant. In general, however, `fold` is useful in many contexts where `reduce` is not (consider the examples given in the `fold` definition – only one of the five folds uses a function with a right identity. The other four may not be performed with reduce).

`(reducse-right f ridentity list)`

`reduce-right` is the `fold-right` variant of reduce. It obeys the following definition:

```
(reduce-right f ridentity '()) = ridentity
(reduce-right f ridentity '(e1)) = (f e1 ridentity) = e1
(reduce-right f ridentity '(e1 e2 ...)) =
    (f e1 (reduce f ridentity (e2 ...)))
```

... in other words, we compute `(fold-right f ridentity list)`.

```
;; Append a bunch of lists together.
;; I.e., (apply append list-of-lists)
(reduce-right append '() list-of-lists)
```

`(unfold p f g seed [tail-gen])`

TODO

`(unfold-right p f g seed [tail-gen])`

TODO

`(map proc list1 ...)`

`proc` is a procedure taking as many arguments as there are list arguments and returning a single value. map applies `proc` element-wise to the elements of the lists and returns a list of the results, in order. The dynamic order in which proc is applied to the elements of the lists is unspecified.

```
(map cadr '((a b) (d e) (g h))) =>  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
    =>  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6)) =>  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b))) =>  (1 2) or (2 1)
```

At least one of the argument lists must be finite:

```
(map + '(3 1 4 1) (circular-list 1 0)) ;; => (4 1 5 1)
```

`(for-each proc clist1 ...)`

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.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)  =>  #(0 1 4 9 16)
```

At least one of the argument lists must be finite.

`(append-map f list ...)`

`(append-map! f list ...)`

Equivalent to:

```
(apply append (map f clist1 clist2 ...))
```

And:

```
(apply append! (map f clist1 clist2 ...))
```

Map `f` over the elements of the lists, just as in the `map` function. However, the results of the applications are appended together to make the final result. `append-map` uses `append` to append the results together; `append-map!` uses `append!`.

The dynamic order in which the various applications of `f` are made is not specified.

Example:

```
(append-map! (lambda (x) (list x (- x))) '(1 3 8)) ;; => (1 -1 3 -3 8 -8)
```

At least one of the list arguments must be finite.

`(map! f list1 ...)`

Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of `list1` to construct the result list.

The dynamic order in which the various applications of `f` are made is not specified. In the n-ary case, `clist2`, `clist3`, ... must have at least as many elements as `list1`.

`(map-in-order f clist1 ...)`

A variant of the map procedure that guarantees to apply `f` across the elements of the `clisti` arguments in a left-to-right order. This is useful for mapping procedures that both have side effects and return useful values.

At least one of the list arguments must be finite.

`(pair-for-each f clist1 ...)`

Like for-each, but `f` is applied to successive sublists of the argument lists. That is, `f` is applied to the `cons` cells of the lists, rather than the lists' elements. These applications occur in left-to-right order.

The f procedure may reliably apply `set-cdr!` to the pairs it is given without altering the sequence of execution.

```
(pair-for-each (lambda (pair) (display pair) (newline)) '(a b c))
 ;; => (a b c)
 ;; => (b c)
 ;; => (c)
```

At least one of the list arguments must be finite.

`(filter-map f clist1 ...)`

Like `map`, but only true values are saved.

```
(filter-map (lambda (x) (and (number? x) (* x x))) '(a 1 b 3 c 7))
    ;; => (1 9 49)
```

The dynamic order in which the various applications of `f` are made is not specified.

At least one of the list arguments must be finite.

Filtering & partitioning

`(filter pred list)`

Return all the elements of list that satisfy predicate `pred`. The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. The returned list may share a common tail with the argument list. The dynamic order in which the various applications of `pred` are made is not specified.

```
(filter even? '(0 7 8 8 43 -4)) => (0 8 8 -4)
```

`(partition pred list)`

Partitions the elements of list with predicate `pred`, and returns two values: the list of in-elements and the list of out-elements. The list is not disordered – elements occur in the result lists in the same order as they occur in the argument list. The dynamic order in which the various applications of pred are made is not specified. One of the returned lists may share a common tail with the argument list.

```
(partition symbol? '(one 2 3 four five 6)) =>
    (one four five)
    (2 3 6)
```

`(remove pred list)`

Returns `list` without the elements that satisfy predicate `pred`:

```
(lambda (pred list) (filter (lambda (x) (not (pred x))) list))
```

The `list` is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. The returned list may share a common tail with the argument list. The dynamic order in which the various applications of `pred` are made is not specified.

```
(remove even? '(0 7 8 8 43 -4)) => (7 43)
```

`(filter! pred list)`

`(partition! pred list)`

`(remove! pred list)`

Linear-update variants of `filter`, `partition` and `remove`. These procedures are allowed, but not required, to alter the cons cells in the argument list to construct the result lists.

Searching

`(find pred clist)`

Return the first element of clist that satisfies predicate `pred`; false if no element does.

```
(find even? '(3 1 4 1 5 9)) => 4
```

Note that `find` has an ambiguity in its lookup semantics – if `find` returns `#f`, you cannot tell (in general) if it found a #f element that satisfied `pred`, or if it did not find any element at all. In many situations, this ambiguity cannot arise – either the list being searched is known not to contain any `#f` elements, or the list is guaranteed to have an element satisfying `pred`. However, in cases where this ambiguity can arise, you should use `find-tail` instead of find – `find-tail` has no such ambiguity.

`(find-tail pred clist)`

Return the first pair of `clist` whose `car` satisfies `pred`. If no pair does, return false.

`find-tail` can be viewed as a general-predicate variant of the member function.

Examples:

```
(find-tail even? '(3 1 37 -8 -5 0 0)) => (-8 -5 0 0)
(find-tail even? '(3 1 37 -5)) => #f

;; MEMBER X LIS:
(find-tail (lambda (elt) (equal? x elt)) lis)
```

In the circular-list case, this procedure “rotates” the list.

`find-tail` is essentially drop-while, where the sense of the predicate is inverted: `find-tail` searches until it finds an element satisfying the predicate; drop-while searches until it finds an element that doesn’t satisfy the predicate.

`(take-while  pred clist)`

`(take-while! pred clist)`

Returns the longest initial prefix of `clist` whose elements all satisfy the predicate `pred`.

`take-while!` is the linear-update variant. It is allowed, but not required, to alter the argument list to produce the result.

```
(take-while even? '(2 18 3 10 22 9)) => (2 18)
```

`(drop-while pred clist)`

Drops the longest initial prefix of `clist` whose elements all satisfy the predicate `pred`, and returns the rest of the list.

```
(drop-while even? '(2 18 3 10 22 9)) => (3 10 22 9)
```

The circular-list case may be viewed as “rotating” the list.

(span pred clist)

(span! pred list)

(break pred clist)

‘(break! pred list)

Span splits the list into the longest initial prefix whose elements all satisfy `pred`, and the remaining tail. Break inverts the sense of the predicate: the tail commences with the first element of the input list that satisfies the predicate.

In other words: `span` finds the intial span of elements satisfying `pred`, and break breaks the list at the first element satisfying `pred`.

Span is equivalent to

```
(values (take-while pred clist)
        (drop-while pred clist))
```

`span!` and `break!` are the linear-update variants. They are allowed, but not required, to alter the argument list to produce the result.

```
(span even? '(2 18 3 10 22 9)) =>
      (2 18)
      (3 10 22 9)

(break even? '(3 1 4 1 5 9)) =>
      (3 1)
      (4 1 5 9)
```

(any pred clist1 ...)

Applies the predicate across the lists, returning true if the predicate returns true on any application.

If there are n list arguments `clist1 ... clistn`, then `pred` must be a procedure taking n arguments and returning a single value, interpreted as a boolean (that is, `#f` means false, and any other value means true).

`any` applies `pred` to the first elements of the `clisti` parameters. If this application returns a true value, any immediately returns that value. Otherwise, it iterates, applying `pred` to the second elements of the `clisti` parameters, then the third, and so forth. The iteration stops when a true value is produced or one of the lists runs out of values; in the latter case, any returns `#f`. The application of pred to the last element of the lists is a tail call.

Note the difference between `find` and `any` – `find` returns the element that satisfied the predicate; `any` returns the true value that the predicate produced.

Like `every`, `any`'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (#t or #f), but a general value.

```
(any integer? '(a 3 b 2.7))   => #t
(any integer? '(a 3.1 b 2.7)) => #f
(any < '(3 1 4 1 5)
       '(2 7 1 8 2)) => #t
```

`(every pred clist1 ...)`

Applies the predicate across the lists, returning true if the predicate returns true on every application.

If there are n list arguments `clist1 ... clistn`, then `pred` must be a procedure taking n arguments and returning a single value, interpreted as a boolean (that is, #f means false, and any other value means true).

`every` applies `pred` to the first elements of the `clisti` parameters. If this application returns false, every immediately returns false. Otherwise, it iterates, applying `pred` to the second elements of the `clisti` parameters, then the third, and so forth. The iteration stops when a false value is produced or one of the lists runs out of values. In the latter case, `every` returns the true value produced by its final application of pred. The application of `pred` to the last element of the lists is a tail call.

If one of the `clisti` has no elements, `every` simply returns #t.

Like `any`, `every`'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (`#t` or `#f`), but a general value.

`(list-index pred clist1 ...)`

Return the index of the leftmost element that satisfies `pred`.

If there are n list arguments `clist1 ... clistn`, then `pred` must be a function taking n arguments and returning a single value, interpreted as a boolean (that is, `#f` means false, and any other value means true).

`list-index` applies `pred` to the first elements of the `clisti` parameters. If this application returns true, `list-index` immediately returns zero. Otherwise, it iterates, applying `pred` to the second elements of the `clisti` parameters, then the third, and so forth. When it finds a tuple of list elements that cause `pred` to return true, it stops and returns the zero-based index of that position in the lists.

The iteration stops when one of the lists runs out of values; in this case, `list-index` returns `#f`.

```
(list-index even? '(3 1 4 1 5 9)) => 2
(list-index < '(3 1 4 1 5 9 2 5 6) '(2 7 1 8 2)) => 1
(list-index = '(3 1 4 1 5 9 2 5 6) '(2 7 1 8 2)) => #f
```

`(member x list [=])`

`(memq x list)`

`(memv x list)`

These procedures return the first sublist of `list` whose `car` is `x`, where the sublists of `list` are the non-empty lists returned by `(drop list i)` for `i` less than the length of list. If `x` does not occur in list, then `#f` is returned. `memq` uses `eq?` to compare `x` with the elements of list, while `memv` uses `eqv?`, and `member` uses `equal?`.

```
(memq 'a '(a b c))          =>  (a b c)
(memq 'b '(a b c))          =>  (b c)
(memq 'a '(b c d))          =>  #f
(memq (list 'a) '(b (a) c)) =>  #f
(member (list 'a) '(b (a) c))         =>  ((a) c)
(memq 101 '(100 101 102))   =>  *unspecified*
(memv 101 '(100 101 102))   =>  (101 102)
```

The comparison procedure is used to compare the elements `ei` of list to the key `x` in this way:

```
(= x ei) ; list is (e1 ... en)
```

That is, the first argument is always `x`, and the second argument is one of the list elements. Thus one can reliably find the first element of list that is greater than five with `(member 5 list <)`

Note that fully general list searching may be performed with the `find-tail` and `find` procedures, e.g.

```
(find-tail even? list) ; Find the first elt with an even key.
```

Deleting

`(delete x list)`

`(delete! x list)`

`delete` uses the comparison procedure `=`, which defaults to `equal?`, to find all elements of list that are equal to `x`, and deletes them from list. The dynamic order in which the various applications of `=` are made is not specified.

The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. The result may share a common tail with the argument list.

Note that fully general element deletion can be performed with the `remove` and `remove!` procedures, e.g.:

```
;; Delete all the even elements from LIS:
(remove even? lis)
```

The comparison procedure is used in this way: `(= x ei)`. That is, `x` is always the first argument, and a list element is always the second argument. The comparison procedure will be used to compare each element of list exactly once; the order in which it is applied to the various `ei` is not specified. Thus, one can reliably remove all the numbers greater than five from a list with `(delete 5 list <)`.

`delete!` is the linear-update variant of `delete`. It is allowed, but not required, to alter the cons cells in its argument list to construct the result.

`(delete-duplicates  list [=])`

`(delete-duplicates! list [=])`

`delete-duplicates` removes duplicate elements from the list argument. If there are multiple equal elements in the argument list, the result list only contains the first or leftmost of these elements in the result. The order of these surviving elements is the same as in the original list – delete-duplicates does not disorder the list (hence it is useful for “cleaning up” association lists).

The `=` parameter is used to compare the elements of the list; it defaults to equal?. If x comes before y in list, then the comparison is performed (= x y). The comparison procedure will be used to compare each pair of elements in list no more than once; the order in which it is applied to the various pairs is not specified.

Implementations of `delete-duplicates` are allowed to share common tails between argument and result lists – for example, if the list argument contains only unique elements, it may simply return exactly this list.

Be aware that, in general, delete-duplicates runs in time O(n2) for n-element lists. Uniquifying long lists can be accomplished in O(n lg n) time by sorting the list to bring equal elements together, then using a linear-time algorithm to remove equal elements. Alternatively, one can use algorithms based on element-marking, with linear-time results.

`delete-duplicates!` is the linear-update variant of `delete-duplicates`; it is allowed, but not required, to alter the cons cells in its argument list to construct the result.

```
(delete-duplicates '(a b a c a b c z)) => (a b c z)

;; Clean up an alist:
(delete-duplicates '((a . 3) (b . 7) (a . 9) (c . 1))
                   (lambda (x y) (eq? (car x) (car y))))
;;  => ((a . 3) (b . 7) (c . 1))
```

Association lists

An "association list" (or "alist") is a list of pairs. The car of each pair contains a key value, and the cdr contains the associated data value. They can be used to construct simple look-up tables in Scheme. Note that association lists are probably inappropriate for performance-critical use on large data; in these cases, hash tables or some other alternative should be employed.

`(assoc key alist [=])`

`(assq key alist)`

`(assv key alist)`

`alist` must be an association list – a list of pairs. These procedures find the first pair in `alist` whose `car` field is `key`, and returns that pair. If no pair in `alist` has `key` as its `car`, then `#f` is returned. `assq` uses `eq?` to compare `key` with the `car` fields of the pairs in `alist`, while `assv` uses `eqv?` and `assoc` uses `equal?`.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                             =>  (a 1)
(assq 'b e)                             =>  (b 2)
(assq 'd e)                             =>  #f
(assq (list 'a) '(((a)) ((b)) ((c))))  =>  #f
(assoc (list 'a) '(((a)) ((b)) ((c)))) =>  ((a))
(assq 5 '((2 3) (5 7) (11 13)))     =>  *unspecified*
(assv 5 '((2 3) (5 7) (11 13)))     =>  (5 7)
```

The comparison procedure is used to compare the elements `ei` of list to the `key` parameter in this way:

```
(= key (car ei)) ; list is (E1 ... En)
```

That is, the first argument is always `key`, and the second argument is one of the list elements. Thus one can reliably find the first entry of alist whose key is greater than five with `(assoc 5 alist <)`

Note that fully general `alist` searching may be performed with the `find-tail` and `find` procedures, e.g.

```
;; Look up the first association in alist with an even key:
(find (lambda (a) (even? (car a))) alist)
```

`(alist-cons key datum alist)`

```
(lambda (key datum alist) (cons (cons key datum) alist))
```

`cons` a new entry mapping `key` to `datum` onto `alist`.

`(alist-copy alist)`

Make a fresh copy of `alist`. This means copying each pair that forms an association as well as the spine of the list, i.e.

```
(lambda (a) (map (lambda (elt) (cons (car elt) (cdr elt))) a))
```

`(alist-delete  key alist [=])`

`(alist-delete! key alist [=])`

`alist-delete` deletes all associations from alist with the given key, using key-comparison procedure `=`, which defaults to equal?. The dynamic order in which the various applications of = are made is not specified.

Return values may share common tails with the alist argument. The alist is not disordered – elements that appear in the result alist occur in the same order as they occur in the argument alist.

The comparison procedure is used to compare the element keys ki of alist's entries to the key parameter in this way: (= key ki). Thus, one can reliably remove all entries of alist whose key is greater than five with (alist-delete 5 alist <)

alist-delete! is the linear-update variant of alist-delete. It is allowed, but not required, to alter cons cells from the alist parameter to construct the result.

Set operations on lists

These procedures implement operations on sets represented as lists of elements. They all take an = argument used to compare elements of lists. This equality procedure is required to be consistent with eq?. That is, it must be the case that `(eq? x y) => (= x y)`.

Note that this implies, in turn, that two lists that are eq? are also set-equal by any legal comparison procedure. This allows for constant-time determination of set operations on eq? lists.

Be aware that these procedures typically run in time O(n * m) for n- and m-element list arguments. Performance-critical applications operating upon large sets will probably wish to use other data structures and algorithms.

`(lset<= = list1 ...)`

Returns true iff every listi is a subset of listi+1, using = for the element-equality procedure. List A is a subset of list B if every element in A is equal to some element of B. When performing an element comparison, the = procedure's first argument is an element of A; its second, an element of B.

```
(lset<= eq? '(a) '(a b a) '(a b c c)) => #t

(lset<= eq?) => #t              ; Trivial cases
(lset<= eq? '(a)) => #t
```

`(lset= = list1 ...)`

Returns true iff every listi is set-equal to listi+1, using = for the element-equality procedure. "Set-equal" simply means that listi is a subset of listi+1, and listi+1 is a subset of listi. The = procedure's first argument is an element of listi; its second is an element of listi+1.

```
(lset= eq? '(b e a) '(a e b) '(e e b a)) => #t

(lset= eq?) => #t               ; Trivial cases
(lset= eq? '(a)) => #t
```

`(lset-adjoin = list elt1 ...)`

Adds the elti elements not already in the list parameter to the result list. The result shares a common tail with the list parameter. The new elements are added to the front of the list, but no guarantees are made about their order. The = parameter is an equality procedure used to determine if an elti is already a member of list. Its first argument is an element of list; its second is one of the elti.

The list parameter is always a suffix of the result – even if the list parameter contains repeated elements, these are not reduced.

```
(lset-adjoin eq? '(a b c d c e) 'a 'e 'i 'o 'u) => (u o i a b c d c e)
```

`(lset-union = list1 ...)`

Returns the union of the lists, using = for the element-equality procedure.

The union of lists A and B is constructed as follows:

- If A is the empty list, the answer is B (or a copy of B).
- Otherwise, the result is initialised to be list A (or a copy of A).
- Proceed through the elements of list B in a left-to-right order. If b is such an element of B, compare every element r of the current result list to b: (= r b). If all comparisons fail, b is consed onto the front of the result.

However, there is no guarantee that = will be applied to every pair of arguments from A and B. In particular, if A is eq? to B, the operation may immediately terminate.

In the n-ary case, the two-argument list-union operation is simply folded across the argument lists.

```
(lset-union eq? '(a b c d e) '(a e i o u)) =>
        (u o i a b c d e)

;; Repeated elements in LIST1 are preserved.
(lset-union eq? '(a a c) '(x a x)) => (x a a c)

;; Trivial cases
(lset-union eq?) => ()
(lset-union eq? '(a b c)) => (a b c)
```

`(lset-intersection = list1 list2 ...)`

Returns the intersection of the lists, using = for the element-equality procedure.

The intersection of lists A and B is comprised of every element of A that is = to some element of B: (= a b), for a in A, and b in B. Note this implies that an element which appears in B and multiple times in list A will also appear multiple times in the result.

The order in which elements appear in the result is the same as they appear in list1 – that is, lset-intersection essentially filters list1, without disarranging element order. The result may share a common tail with list1.

In the n-ary case, the two-argument list-intersection operation is simply folded across the argument lists. However, the dynamic order in which the applications of = are made is not specified. The procedure may check an element of list1 for membership in every other list before proceeding to consider the next element of list1, or it may completely intersect list1 and list2 before proceeding to list3, or it may go about its work in some third order.

```
(lset-intersection eq? '(a b c d e) '(a e i o u)) => (a e)

;; Repeated elements in LIST1 are preserved.
(lset-intersection eq? '(a x y a) '(x a x z)) => '(a x a)

(lset-intersection eq? '(a b c)) => (a b c)     ; Trivial case
```

(lset-difference = list1 list2 ...)

Returns the difference of the lists, using = for the element-equality procedure – all the elements of list1 that are not = to any element from one of the other listi parameters.

The = procedure's first argument is always an element of list1; its second is an element of one of the other listi. Elements that are repeated multiple times in the list1 parameter will occur multiple times in the result. The order in which elements appear in the result is the same as they appear in list1 – that is, lset-difference essentially filters list1, without disarranging element order. The result may share a common tail with list1. The dynamic order in which the applications of = are made is not specified. The procedure may check an element of list1 for membership in every other list before proceeding to consider the next element of list1, or it may completely compute the difference of list1 and list2 before proceeding to list3, or it may go about its work in some third order.

```
(lset-difference eq? '(a b c d e) '(a e i o u)) => (b c d)

(lset-difference eq? '(a b c)) => (a b c) ; Trivial case
```

'(lset-xor = list1 . . .)

Returns the exclusive-or of the sets, using = for the element-equality procedure. If there are exactly two lists, this is all the elements that appear in exactly one of the two lists. The operation is associative, and thus extends to the n-ary case – the elements that appear in an odd number of the lists. The result may share a common tail with any of the listi parameters.

More precisely, for two lists A and B, A xor B is a list of

- every element a of A such that there is no element b of B such that (= a b), and
- every element b of B such that there is no element a of A such that (= b a).

However, an implementation is allowed to assume that = is symmetric – that is, that

```
(= a b) => (= b a).
```

This means, for example, that if a comparison (= a b) produces true for some a in A and b in B, both a and b may be removed from inclusion in the result.

In the n-ary case, the binary-xor operation is simply folded across the lists.

```
(lset-xor eq? '(a b c d e) '(a e i o u)) => (d c b i o u)

;; Trivial cases.
(lset-xor eq?) => ()
(lset-xor eq? '(a b c d e)) => (a b c d e)
```

`(lset-diff+intersection = list1 list2 ...)`

Returns two values – the difference and the intersection of the lists. Is equivalent to:

```
(values (lset-difference = list1 list2 ...)
        (lset-intersection = list1
                             (lset-union = list2 ...)))
```

But can be implemented more efficiently.

The = procedure's first argument is an element of list1; its second is an element of one of the other listi.

Either of the answer lists may share a common tail with list1. This operation essentially partitions list1.

`(lset-union! list1 ...)`

`(lset-intersection! list1 ...)`

`(lset-difference! list1 ...)`

`(lset-xor! list1 ...)`

`(lset-diff+intersection! list1 ...)`

These are linear-update variants. They are allowed, but not required, to use the cons cells in their first list parameter to construct their answer. lset-union! is permitted to recycle cons cells from any of its list arguments.

Primitive side-effects

(set-car! pair object)

(set-cdr! pair object)

These procedures store object in the car and cdr field of pair, respectively. The value returned is unspecified.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3) =>  *unspecified*
(set-car! (g) 3) =>  *error*
```

(srfi srfi-2)

This is based on SRFI-2.

Abstract

Like an ordinary `and`, an `and-let*` special form evaluates its arguments – expressions – one after another in order, till the first one that yields `#f`. Unlike `and`, however, a non-`#f` result of one expression can be bound to a fresh variable and used in the subsequent expressions. `and-let*` is a cross-breed between `let*` and `and`.

Reference

and-let*

`and-let*` is a generalized `and`: it evaluates a sequence of forms one after another till the first one that yields `#f`; the non-`#f` result of a form can be bound to a fresh variable and used in the subsequent forms.

(srfi srfi-4)

This is based on SRFI-4.

Abstract

This SRFI describes a set of datatypes for vectors whose elements are of the same numeric type (signed or unsigned exact integer or inexact real of a given precision). These datatypes support operations analogous to the Scheme vector type, but they are distinct datatypes.

Reference

Signed 8 bits integer

s8vector?

make-s8vector

s8vector

s8vector-length

s8vector-ref

s8vector-set!

s8vector->list

list->s8vector

Signed 16 bits integer

s16vector?

make-s16vector

s16vector

s16vector-length

s16vector-ref

s16vector-set!

s16vector->list

list->s16vector

Signed 32 bits integer

s32vector?

make-s32vector

s32vector

s32vector-length

s32vector-ref

s32vector-set!

s32vector->list

list->s32vector

Signed 64 bits integer

s64vector?

make-s64vector

s64vector

s64vector-length

s64vector-ref

s64vector-set!

s64vector->list

list->s64vector

Unsigned 8 bits integer

u8vector?

make-u8vector

u8vector

u8vector-length

u8vector-ref

u8vector-set!

u8vector->list

list->u8vector

Unsigned 16 bits integer

u16vector?

make-u16vector

u16vector

u16vector-length

u16vector-ref

u16vector-set!

u16vector->list

list->u16vector

Unsigned 32 bits integer

u32vector?

make-u32vector

u32vector

u32vector-length

u32vector-ref

u32vector-set!

u32vector->list

list->u32vector

Unsigned 64 bits integer

u64vector?

make-u64vector

u64vector

u64vector-length

u64vector-ref

u64vector-set!

u64vector->list

list->u64vector

32 bits float

f32vector?

make-f32vector

f32vector

f32vector-length

f32vector-ref

f32vector-set!

f32vector->list

list->f32vector

64 bits float

f64vector?

make-f64vector

f64vector

f64vector-length

f64vector-ref

f64vector-set!

f64vector->list

list->f64vector

`(srfi srfi-5)`

This is based on SRFI-5.

Abstract

The named-`let` incarnation of the `let` form has two slight inconsistencies with the `define` form. As defined, the `let` form makes no accommodation for rest arguments, an issue of functionality and consistency. As defined, the let form does not accommodate signature-style syntax, an issue of aesthetics and consistency. Both issues are addressed here in a manner which is compatible with the traditional `let` form but for minor extensions.

Reference

TODO

`(srfi srfi-6)`

This is based on SRFI-6.

Abstract

Scheme's i/o primitives are extended by adding three new procedures that

- create an input port from a string,
- create an output port whose contents are accumulated in Scheme's working memory instead of an external file, and
- extract the accumulated contents of an in-memory output port and return them in the form of a string.

Reference

(open-input-string string)

Takes a string and returns an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

```
(define p
  (open-input-string "(a . (b . (c . ()))) 34"))

(input-port? p) ;; =>  #t
(read p) ;; => (a b c)
(read p) ;; => 34
(eof-object? (peek-char p)) ;; => #t
```

(open-output-string)

Returns an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

```
(let ((q (open-output-string))
      (x '(a b c)))
  (write (car x) q)
  (write (cdr x) q)
  (get-output-string q)) ;; => "a(b c)"
```

(get-output-string output-port)

Given an output port created by `open-output-string`, returns a string consisting of the characters that have been output to the port so far.

(srfi srfi-8)

This is based on SRFI-8.

Abstract

The only mechanism that R5RS provides for binding identifiers to the values of a multiple-valued expression is the primitive `call-with-values`. This SRFI proposes a more concise, more readable syntax for creating such bindings.

Reference

`(receive <formals> <expression> <body>)` syntax

(srfi srfi-9)

This is based on SRFI-9.

Abstract

Syntax for creating new data types, called record types. A predicate, constructor, and field accessors and modifiers are defined for each record type. Each new record type is distinct from all existing types, including other record types and Scheme's predefined types.

Reference

`(define-record-type ...)` syntax

The following:

```
(define-record-type <pare>
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

Defines `KONS` to be a constructor, `KAR` and `KDR` to be accessors, `SET-KAR!` to be a modifier, and `PARE?` to be a predicate for `<PAREs>`.

```
(pare? (kons 1 2))        --> #t
(pare? (cons 1 2))        --> #f
(kar (kons 1 2))          --> 1
(kdr (kons 1 2))          --> 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k))                --> 3
```

(srfi srfi-13)

This is based on SRFI-13.

Abstract

TODO

Reference

Predicates

`string?`

`string-null?`

`string-every`

`string-any`

Constructors

`make-string`

`string`

`string-tabulate`

List & string conversion

`string->list`

`list->string`

`reverse-list->string`

`string-join`

Selection

`string-length`

`string-ref`

`string-copy`

`substring/shared`

`string-copy!`

`string-take`

`string-take-right`

`string-drop`

`string-drop-right`

`string-pad`

`string-pad-right`

`string-trim`

`string-trim-right`

`string-trim-both`

Modification

`string-set!`

`string-fill!`

Comparison

string-compare

string-compare-ci

string<>

string=

string<

string>

string<=

string>=

string-ci<>

string-ci=

string-ci<

string-ci>

string-ci<=

string-ci>=

string-hash

string-hash-ci

Prefixes & suffixes

string-prefix-length

`string-suffix-length`

`string-prefix-length-ci`

`string-suffix-length-ci`

`string-prefix?`

`string-suffix?`

`string-prefix-ci?`

`string-suffix-ci?`

Searching

`string-index`

`string-index-right`

`string-skip`

`string-skip-right`

`string-count`

`string-contains`

`string-contains-ci`

Alphabetic case mapping

`string-titlecase`

`string-upcase`

`string-downcase`

`string-titlecase!`

`string-upcase!`

`string-downcase!`

Reverse & append

`string-reverse`

`string-reverse!`

`string-append`

`string-concatenate`

`string-concatenate/shared`

`string-append/shared`

`string-concatenate-reverse`

`string-concatenate-reverse/shared`

Fold, unfold & map

`string-map`

`string-map!`

`string-fold`

`string-fold-right`

`string-unfold`

`string-unfold-right`

`string-for-each`

`string-for-each-index`

Replicate & rotate

`xsubstring`

`string-xcopy!`

Miscellaneous: insertion, parsing

`string-replace`

`string-tokenize`

Filtering & deleting

`string-filter`

`string-delete`

Low-level procedures

`string-parse-start+end`

`string-parse-final-start+end`

`let-string-start+end`

`check-substring-spec`

`substring-spec-ok?`

`make-kmp-restart-vector`

`kmp-step`

`string-kmp-partial-search`

`(srfi srfi-14)`

This library is based on SRFI-14.

Abstract

The ability to efficiently represent and manipulate sets of characters is an unglamorous but very useful capability for text-processing code – one that tends to pop up in the definitions of other libraries.

Reference

`->char-set`

TODO

`char-set`

TODO

`char-set->list`

TODO

`char-set->string`

TODO

`char-set-adjoin`

TODO

`char-set-adjoin!`

TODO

`char-set-any`

TODO

`char-set-complement`

TODO

`char-set-complement!`

TODO

`char-set-contains?`

TODO

`char-set-copy`

TODO

`char-set-count`

TODO

`char-set-cursor`

TODO

`char-set-cursor-next`

TODO

`char-set-delete`

TODO

`char-set-delete!`

TODO

`char-set-diff+intersection`

TODO

`char-set-diff+intersection!`

TODO

`char-set-difference`

TODO

`char-set-difference!`

TODO

`char-set-every`

TODO

`char-set-filter`

TODO

`char-set-filter!`

TODO

`char-set-fold`

TODO

`char-set-for-each`

TODO

`char-set-hash`

TODO

`char-set-intersection`

TODO

`char-set-intersection!`

TODO

`char-set-map`

TODO

`char-set-ref`

TODO

`char-set-size`

TODO

`char-set-unfold`

TODO

`char-set-unfold!`

TODO

`char-set-union`

TODO

`char-set-union!`

TODO

`char-set-xor`

TODO

`char-set-xor!`

TODO

`char-set:ascii`

TODO

`char-set:blank`

TODO

`char-set:digit`

TODO

`char-set:empty`

TODO

`char-set:full`

TODO

`char-set:graphic`

TODO

`char-set:hex-digit`

TODO

`char-set:iso-control`

TODO

`char-set:letter`

TODO

`char-set:letter+digit`

TODO

`char-set:lower-case`

TODO

`char-set:printing`

TODO

`char-set:punctuation`

TODO

`char-set:symbol`

TODO

`char-set:title-case`

TODO

`char-set:upper-case`

TODO

`char-set:whitespace`

TODO

`char-set<=`

TODO

`char-set=`

TODO

`char-set?`

TODO

`end-of-char-set?`

TODO

`list->char-set`

TODO

`list->char-set!`

TODO

`string->char-set`

TODO

`string->char-set!`

TODO

`ucs-range->char-set`

TODO

`ucs-range->char-set!`

TODO

`(srfi srfi-16)`

`(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‘.

Example:

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

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

(srfi srfi-17)

This is based on SRFI-17.

Abstract

Allow procedure calls that evaluate to the "value of a location" to be used to set the value of the location, when used as the first operand of `set!`. For example:

```
(set! (car x) (car y))
```

becomes equivalent to

```
(set-car! x (car y))
```

Reference

```
(set! (car x) v) == (set-car! x v)
(set! (cdr x) v) == (set-cdr! x v)
(set! (caar x) v) == (set-car! (car x) v)
(set! (cadr x) v) == (set-car! (cdr x) v)
....
(set! (caXXr x) v) == (set-car! (cXXr x) v)
(set! (cdXXr x) v) == (set-cdr! (cXXr x) v)
(set! (string-ref x i) v) == (string-set! x i v)
(set! (vector-ref x i) v) == (vector-set! x i v)
```

(srfi srfi-19)

This is based on SRFI-19.

Abstract

TODO

Reference

`time-duration`

`time-monotonic`

`time-process`

`time-tai`

`time-thread`

`time-utc`

`current-date`

`current-julian-day`

`current-modified-julian-day`

`current-time`

`time-resolution`

`make-time`

`time?`

`time-type`

`time-nanosecond`

`time-second`

`set-time-type!`

`set-time-nanosecond!`

`set-time-second!`

`copy-time`

`time<=?`

```
time<?

time=?

time>=?

time>?

time-difference

time-difference!

add-duration

add-duration!

subtract-duration

subtract-duration!

make-date

date?

date-nanosecond

date-second

date-minute

date-hour

date-day

date-month
```

```
date-year

date-zone-offset

date-year-day

date-week-day

date-week-number

date->julian-day

date->modified-julian-day

date->time-monotonic

date->time-tai

date->time-utc

julian-day->date

julian-day->time-monotonic

julian-day->time-tai

julian-day->time-utc

modified-julian-day->date

modified-julian-day->time-monotonic

modified-julian-day->time-tai

modified-julian-day->time-utc
```

`time-monotonic->date`

`time-monotonic->julian-day`

`time-monotonic->modified-julian-day`

`time-monotonic->time-tai`

`time-monotonic->time-tai!`

`time-monotonic->time-utc`

`time-monotonic->time-utc!`

`time-tai->date`

`time-tai->julian-day`

`time-tai->modified-julian-day`

`time-tai->time-monotonic`

`time-tai->time-monotonic!`

`time-tai->time-utc`

`time-tai->time-utc!`

`time-utc->date`

`time-utc->julian-day`

`time-utc->modified-julian-day`

`time-utc->time-monotonic`

`time-utc->time-monotonic!`

`time-utc->time-tai`

`time-utc->time-tai!`

`date->string`

`string->date`

(srfi srfi-23)

This is based on SRFI-23.

Abstract

A mechanism is proposed to allow Scheme code to report errors and abort execution. The proposed mechanism is already implemented in several Scheme systems and can be implemented, albeit imperfectly, in any R5RS conforming Scheme.

Reference

```
(error <reason> [arg ...])
```

(srfi srfi-25)

This is based on SRFI-25.

Abstract

A core set of procedures for creating and manipulating heterogeneous multidimensional arrays is proposed. The design is consistent with the rest of Scheme and independent of other container data types. It provides easy sharing of parts of an array as other arrays without copying, encouraging a declarative style of programming.

Reference

TODO

`(srfi srfi-26)`

This is based on SRFI-26.

Abstract

When programming in functional style, it is frequently necessary to specialize some of the parameters of a multi-parameter procedure. For example, from the binary operation cons one might want to obtain the unary operation (lambda (x) (cons 1 x)). This specialization of parameters is also known as “partial application”, “operator section” or “projection”.

Reference

`(cut ...)` syntax

`(cute ...)` syntax

`(srfi srfi-28)`

This is based on SRFI-28.

Abstract

A method of interpreting a Scheme string which contains a number of escape sequences that are replaced with other string data according to the semantics of each sequence. Also called string interpolation.

Reference

`(format format-string [obj ...])`

Accepts a message template (a Scheme string), and processes it, replacing any escape sequences in order with one or more characters, the characters themselves dependent on the semantics of the escape sequence encountered.

An escape sequence is a two character sequence in the string where the first character is a tilde ‘~’. Each escape code’s meaning is as follows:

- `~a` The corresponding value is inserted into the string as if printed with display.
- `~s` The corresponding value is inserted into the string as if printed with write.
- `~%` A newline is inserted.
- `~~` A tilde `~` is inserted.

`~a` and `~s`, when encountered, require a corresponding Scheme value to be present after the format string. The values provided as operands are used by the escape sequences in order. It is an error if fewer values are provided than escape sequences that require them.

`~%` and `~~` require no corresponding value.

Examples:

```
(format "Hello, ~a" "World!")
;; => "Hello, World!"

(format "Error, list is too short: ~s~%" '(one "two" 3))
;; => "Error, list is too short: (one \"two\" 3))"
```

(srfi srfi-29)

This is based on SRFI-29.

Abstract

An interface to retrieving and displaying locale sensitive messages. A Scheme program can register one or more translations of templated messages, and then write Scheme code that can transparently retrieve the appropriate message for the locale under which the Scheme system is running.

Reference

`current-language`

`current-country`

`current-locale-details`

`declare-bundle!`

`store-bundle`

`store-bundle!`

`load-bundle!`

`localized-template`

`(srfi srfi-31)`

This is based on SRFI-31.

Abstract

TODO

Reference

TODO

`(srfi srfi-34)`

This is based on SRFI-34.

Abstract

TODO

Reference

`with-exception-handler`

`guard`

`raise`

(srfi srfi-35)

This is based on SRFI-35.

Abstract

Defines constructs for creating and inspecting condition types and values. A condition value encapsulates information about an exceptional situation, or exception. This SRFI also defines a few basic condition types.

Reference

`make-condition-type`

`condition-type?`

`condition-has-type?`

`condition-ref`

`make-compound-condition`

`extract-condition`

`define-condition-type`

`&condition`

`make-condition`

`condition?`

`condition`

`&serious`

serious-condition?

&error

error?

&message

message-condition?

condition-message

(srfi srfi-37)

This is based on SRFI-37.

Abstract

Many operating systems make the set of argument strings used to invoke a program available (often following the program name string in an array called argv). Most programs need to parse and process these argument strings in one way or another. This SRFI describes a set of procedures that support processing program arguments according to POSIX and GNU C Library Reference Manual guidelines.

Reference

args-fold

option

option?

option-names

option-required-arg?

option-optional-arg?

option-processor

(srfi srfi-38)

This is based on SRFI-38.

Abstract

TODO

Reference

(write-with-shared-structure obj [port [optarg]])

(read-with-shared-structure [port])

(srfi srfi-39)

This is based on SRFI-39.

Abstract

This SRFI defines parameter objects, the procedure make-parameter to create parameter objects and the parameterize special form to dynamically bind parameter objects. In the dynamic environment, each parameter object is bound to a cell containing the value of the parameter. When a procedure is called the called procedure inherits the dynamic environment from the caller. The parameterize special form allows the binding of a parameter object to be changed for the dynamic extent of its body.

Reference

make-parameter

parameterize

`(srfi srfi-41)`

This is based on SRFI-41.

Abstract

Streams, sometimes called lazy lists, are a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again.

Reference

`stream-null`

`stream-cons`

`stream?`

`stream-null?`

`stream-pair?`

`stream-car`

`stream-cdr`

`stream-lambda`

`define-stream`

`list->stream`

`port->stream`

`stream`

stream->list

stream-append

stream-concat

stream-constant

stream-drop

stream-drop-while

stream-filter

stream-fold

stream-for-each

stream-from

stream-iterate

stream-length

stream-let

stream-map

stream-match

stream-of

stream-range

stream-ref

`stream-reverse`

`stream-scan`

`stream-take`

`stream-take-while`

`stream-unfold`

`stream-unfolds`

`stream-zip`

`(srfi srfi-42)`

This is based on SRFI-42.

Abstract

TODO

Reference

`do-ec`

`list-ec`

`append-ec`

`string-ec`

`string-append-ec`

`vector-ec`

```
vector-of-length-ec

sum-ec

product-ec

min-ec

max-ec

any?-ec

every?-ec

first-ec

last-ec

fold-ec

fold3-ec

:

:list

:string

:vector

:integers

:range

:real-range
```

`:char-range`

`:port`

`:dispatched`

`:do`

`:let`

`:parallel`

`:while`

`:until`

`:-dispatch-ref`

`:-dispatch-set!`

`make-initial-:-dispatch`

`dispatch-union`

`:generator-proc`

(srfi srfi-43)

This is based on SRFI-43.

Abstract

TODO

Reference

TODO

`(srfi srfi-45)`

This is based on SRFI-45.

Abstract

TODO

Reference

`delay`

`lazy`

`force`

`eager`

`(srfi srfi-48)`

This is based on SRFI-48.

Abstract

TODO

Reference

TODO

`(srfi srfi-51)`

This is based on SRFI-51.

Abstract

TODO

Reference

TODO

`(srfi srfi-54)`

This is based on SRFI-54.

Abstract

TODO

Reference

TODO

`(srfi srfi-60)`

This is based on SRFI-60.

Abstract

TODO

Reference

logand

bitwise-and

logior

bitwise-ior

logxor

bitwise-xor

lognot

bitwise-not

bitwise-if

bitwise-merge

logtest

any-bits-set?

logcount

bit-count

integer-length

log2-binary-factors

first-set-bit

logbit?

bit-set?

copy-bit

bit-field

copy-bit-field

ash

arithmetic-shift

rotate-bit-field

reverse-bit-field

integer->list

integer->list

list->integer

booleans->integer

`(srfi srfi-61)`

This is based on SRFI-61.

Abstract

TODO

Reference

TODO

`(srfi srfi-67)`

This is based on SRFI-67.

Abstract

TODO

Reference

TODO

(srfi srfi-69)

This is based on SRFI-69.

Abstract

TODO

Reference

TODO

(srfi srfi-98)

This is based on SRFI-98.

Abstract

TODO

Reference

TODO

(srfi srfi-99)

This is based on SRFI-99.

Abstract

TODO

Reference

TODO

`(srfi srfi-101)`

This library is based on SRFI-101.

Abstract

TODO

Reference

TODO

`(srfi srfi-111)`

This library is based on SRFI-111.

Abstract

Boxes are objects with a single mutable state. Several Schemes have them, sometimes called cells. A constructor, predicate, accessor, and mutator are provided.

Reference

`(box value)`

Constructor. Returns a newly allocated box initialized to value.

`(box? object)`

Predicate. Returns `#t` if object is a box, and `#f` otherwise.

`(unbox box)`

Accessor. Returns the current value of box.

`(set-box! box value)`

Mutator. Changes box to hold value.

`(srfi srfi-113)`

This library is based on SRFI-113.

Abstract

Sets and bags (also known as multisets) are unordered collections that can contain any Scheme object. Sets enforce the constraint that no two elements can be the same in the sense of the set's associated equality predicate; bags do not.

Reference

TODO

Set

set

set-unfold

set?

set-contains?

set-empty?

set-disjoint?

set-member

set-element-comparator

set-adjoin

set-adjoin!

set-replace

set-replace!

set-delete

set-delete!

set-delete-all

set-delete-all!

set-search!

set-size

set-find

set-count

set-any?

set-every?

set-map

set-for-each

set-fold

set-filter

set-remove

set-remove

set-partition

set-filter!

set-remove!

set-partition!

set-copy

set->list

list->set

list->set!

set=?

set<?

set>?

set<=?

set>=?

set-union

set-intersection

set-difference

set-xor

set-union!

set-intersection!

set-difference!

set-xor!

set-comparator

Bag

bag

bag-unfold

bag?

bag-contains?

bag-empty?

bag-disjoint?

bag-member

bag-element-comparator

bag-adjoin

bag-adjoin!

bag-replace

bag-replace!

bag-delete

bag-delete!

bag-delete-all

bag-delete-all!

bag-search!

bag-size

bag-find

bag-count

bag-any?

bag-every?

bag-map

bag-for-each

bag-fold

bag-filter

bag-remove

bag-partition

bag-filter!

bag-remove!

bag-partition!

bag-copy

bag->list

list->bag

list->bag!

bag=?

bag<?

bag>?

bag<=?

bag>=?

bag-union

bag-intersection

bag-difference

bag-xor

bag-union!

bag-intersection!

bag-difference!

bag-xor!

bag-comparator

bag-sum

bag-sum!

bag-product

bag-product!

bag-unique-size

bag-element-count

bag-for-each-unique

bag-fold-unique

bag-increment!

bag-decrement!

bag->set

set->bag

set->bag!

bag->alist

alist->bag

`(srfi srfi-115)`

This library is based on SRFI-115.

Abstract

TODO

Reference

TODO

(srfi srfi-116)

This library is based on SRFI-116.

Abstract

TODO

Reference

TODO

(srfi srfi-117)

This library is based on SRFI-117.

Abstract

TODO

Reference

TODO

(srfi srfi-124)

This library is based on SRFI-124.

Abstract

TODO

Reference

`(ephemeron? obj)`

`make-ephemeron`

`ephemeron-broken?`

`ephemeron-key`

`ephemeron-value`

`(srfi srfi-125)`

This library is based on srfi-125.

The library doesn't implement deprecated features. Application must rely on `(scheme comparator)` to specify equal predicate and hash function.

Abstract

This SRFI defines an interface to hash tables, which are widely recognized as a fundamental data structure for a wide variety of applications. A hash table is a data structure that:

- Is disjoint from all other types.
- Provides a mapping from objects known as keys to corresponding objects known as values.
- Keys may be any Scheme objects in some kinds of hash tables, but are restricted in other kinds.
- Values may be any Scheme objects.
- Has no intrinsic order for the key-value associations it contains.
- Provides an equality predicate which defines when a proposed key is the same as an existing key. No table may contain more than one value for a given key.
- Provides a hash function which maps a candidate key into a non-negative exact integer.
- Supports mutation as the primary means of setting the contents of a table.

- Provides key lookup and destructive update in (expected) amortized constant time, provided a satisfactory hash function is available.
- Does not guarantee that whole-table operations work in the presence of concurrent mutation of the whole hash table (values may be safely mutated).

Reference

Constructors

`(make-hash-table comparator . args)`

Returns a newly allocated hash table using `(scheme comparator)` object `COMPARATOR`. For the time being, `ARGS` is ignored.

`(hash-table comparator [key value] ...)`

Returns a newly allocated hash table using `(scheme comparator)` object `COMPARATOR`. For each pair of arguments, an association is added to the new hash table with key as its key and value as its value. If the same key (in the sense of the equality predicate) is specified more than once, it is an error.

`(hash-table-unfold stop? mapper successor seed comparator args ...)`

Create a new hash table as if by `make-hash-table` using `comparator` and the `args`. If the result of applying the predicate `stop?` to `seed` is true, return the hash table. Otherwise, apply the procedure `mapper` to `seed`. `mapper` returns two values, which are inserted into the hash table as the key and the value respectively. Then get a new `seed` by applying the procedure `successor` to `seed`, and repeat this algorithm.

`(alist->hash-table alist comparator arg ...)`

Returns a newly allocated hash-table as if by `make-hash-table` using `comparator` and the `args`. It is then initialized from the associations of `alist`. Associations earlier in the list take precedence over those that come later.

Predicates

`(hash-table? obj)`

Returns #t if obj is a hash table, and #f otherwise

`(hash-table-contains? hash-table key)`

Returns #t if there is any association to key in hash-table, and #f otherwise.

`(hash-table-empty? hash-table)`

Returns #t if hash-table contains no associations, and #f otherwise.

`(hash-table=? value-comparator hash-table1 hash-table2)`

Returns #t if hash-table1 and hash-table2 have the same keys (in the sense of their common equality predicate) and each key has the same value (in the sense of value-comparator), and #f otherwise.

`(hash-table-mutable? hash-table)`

Returns #t if the hash table is mutable.

Accessors

The following procedures, given a key, return the corresponding value.

`(hash-table-ref hash-table key [failure [success]])`

Extracts the value associated to key in hash-table, invokes the procedure success on it, and returns its result; if success is not provided, then the value itself is returned. If key is not contained in hash-table and failure is supplied, then failure is invoked on no arguments and its result is returned.

`(hash-table-ref/default hash-table key default)`

TODO

Mutators

The following procedures alter the associations in a hash table either unconditionally, or conditionally on the presence or absence of a specified key. It is an error to add an association to a hash table whose key does not satisfy the type test predicate of the comparator used to create the hash table.

`(hash-table-set! hash-table key value ...)`

Repeatedly mutates hash-table, creating new associations in it by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, it is deleted. It is an error if the type check procedure of the comparator of hash-table, when invoked on a key, does not return #t. Likewise, it is an error if a key is not a valid argument to the equality predicate of hash-table. Returns an unspecified value.

`(hash-table-delete! hash-table key ...)`

Deletes any association to each key in hash-table and returns the number of keys that had associations.

`(hash-table-intern! hash-table key failure)`

Effectively invokes hash-table-ref with the given arguments and returns what it returns. If key was not found in hash-table, its value is set to the result of calling failure.

`(hash-table-update! hash-table key updater [failure [success]])`

TODO:

`(hash-table-pop! hash-table)`

Chooses an arbitrary association from hash-table and removes it, returning the key and value as two values.

It is an error if hash-table is empty.

`(hash-table-clear! hash-table)`

Delete all the associations from hash-table.

The whole hash table

These procedures process the associations of the hash table in an unspecified order.

`(hash-table-size hash-table)`

Returns the number of associations in hash-table as an exact integer.

`(hash-table-keys hash-table)`

Returns a newly allocated list of all the keys in hash-table.

`(hash-table-values hash-table)`

Returns a newly allocated list of all the keys in hash-table.

`(hash-table-entries hash-table)`

Returns two values, a newly allocated list of all the keys in hash-table and a newly allocated list of all the values in hash-table in the corresponding order.

`(hash-table-find proc hash-table failure)`

For each association of hash-table, invoke proc on its key and value. If proc returns true, then hash-table-find returns what proc returns. If all the calls to proc return #f, return the result of invoking the thunk failure.

`(hash-table-count pred hash-table)`

For each association of hash-table, invoke pred on its key and value. Return the number of calls to pred which returned true.

Mapping and folding

These procedures process the associations of the hash table in an unspecified order.

`(hash-table-map proc comparator hash-table)`

Returns a newly allocated hash table as if by `(make-hash-table comparator)`. Calls `PROC` for every association in `hash-table` with the value of the association. The key of the association and the result of invoking `proc` are entered into the new hash table. Note that this is not the result of lifting mapping over the domain of hash tables, but it is considered more useful.

If comparator recognizes multiple keys in the hash-table as equivalent, any one of such associations is taken.

`(hash-table-for-each proc hash-table)`

Calls proc for every association in hash-table with two arguments: the key of the association and the value of the association. The value returned by proc is discarded. Returns an unspecified value.

`(hash-table-map! proc hash-table)`

Calls proc for every association in hash-table with two arguments: the key of the association and the value of the association. The value returned by proc is used to update the value of the association. Returns an unspecified value.

`(hash-table-map->list proc hash-table)`

Calls proc for every association in hash-table with two arguments: the key of the association and the value of the association. The values returned by the invocations of proc are accumulated into a list, which is returned.

`(hash-table-fold proc seed hash-table)`

Calls proc for every association in hash-table with three arguments: the key of the association, the value of the association, and an accumulated value val. Val is seed for the first invocation of procedure, and for subsequent invocations of proc, the returned value of the previous invocation. The value returned by hash-table-fold is the return value of the last invocation of proc.

`(hash-table-prune! proc hash-table)`

Calls proc for every association in hash-table with two arguments, the key and the value of the association, and removes all associations from hash-table for which proc returns true. Returns an unspecified value.

Copying and conversion

`(hash-table-copy hash-table [mutable?])`

Returns a newly allocated hash table with the same properties and associations as hash-table. If the second argument is present and is true, the new hash table is mutable. Otherwise it is immutable provided that the implementation supports immutable hash tables.

`(hash-table-empty-copy hash-table)`

Returns a newly allocated mutable hash table with the same properties as hash-table, but with no associations.

`(hash-table->alist hash-table)`

Returns an alist with the same associations as hash-table in an unspecified order.

Hash tables as sets

`(hash-table-union! hash-table1 hash-table2)`

Adds the associations of hash-table2 to hash-table1 and returns hash-table1. If a key appears in both hash tables, its value is set to the value appearing in hash-table1. Returns hash-table1.

`(hash-table-intersection! hash-table1 hash-table2)`

Deletes the associations from hash-table1 whose keys don't also appear in hash-table2 and returns hash-table1.

`(hash-table-difference! hash-table1 hash-table2)`

Deletes the associations of hash-table1 whose keys are also present in hash-table2 and returns hash-table1.

`(hash-table-xor! hash-table1 hash-table2)`

Deletes the associations of hash-table1 whose keys are also present in hash-table2, and then adds the associations of hash-table2 whose keys are not present in hash-table1 to hash-table1. Returns hash-table1.

`(srfi srfi-127)`

This library is based on SRFI-127.

Abstract

TODO

Reference

TODO

`(srfi srfi-128)`

This library is based on SRFI-128.

Abstract

A comparator is an object of a disjoint type. It is a bundle of procedures that are useful for comparing two objects either for equality or for ordering. There are four procedures in the bundle:

- The type test predicate returns #t if its argument has the correct type to be passed as an argument to the other three procedures, and #f otherwise.
- The equality predicate returns #t if the two objects are the same in the sense of the comparator, and #f otherwise. It is the programmer's responsibility to ensure that it is reflexive, symmetric, transitive, and can handle any arguments that satisfy the type test predicate.
- The comparison procedure returns -1, 0, or 1 if the first object precedes the second, is equal to the second, or follows the second, respectively, in a total order defined by the comparator. It is the programmer's responsibility to ensure that it is reflexive, weakly antisymmetric, transitive, can handle any arguments that satisfy the type test predicate, and returns 0 iff the equality predicate returns #t.
- The hash function takes one argument, and returns an exact non-negative integer. It is the programmer's responsibility to ensure that it can handle any argument that satisfies the type test predicate, and that it returns the same value on two objects if the equality predicate says they are the same (but not necessarily the converse).

It is also the programmer's responsibility to ensure that all four procedures provide the same result whenever they are applied to the same object(s) (in the sense of eqv?), unless the object(s) have been mutated since the last invocation. In particular, they must not depend in any way on memory addresses in implementations where the garbage collector can move objects in memory.

Limitations

The comparator objects defined in this library are not applicable to circular structure or to NaNs or objects containing them. Attempts to pass any such objects to any procedure defined here, or to any procedure that is part of a comparator defined here, is an error except as otherwise noted.

Reference

Predicates

`(comparator? obj)`

Returns #t if obj is a comparator, and #f otherwise.

(comparator-comparison-procedure? comparator)

Returns #t if comparator has a supplied comparison procedure, and #f otherwise.

(comparator-hash-function? comparator)

Returns #t if comparator has a supplied hash function, and #f otherwise.

Standard comparators

boolean-comparator

Compares booleans using the total order #f < #t.

char-comparator

Compares characters using the total order implied by char<?. On R6RS and R7RS systems, this is Unicode codepoint order.

char-ci-comparator

Compares characters using the total order implied by char-ci<? On R6RS and R7RS systems, this is Unicode codepoint order after the characters have been folded to lower case.

string-comparator

Compares strings using the total order implied by string<?. Note that this order is implementation-dependent.

string-ci-comparator

Compares strings using the total order implied by string-ci<?. Note that this order is implementation-dependent.

symbol-comparator

Compares symbols using the total order implied by applying symbol->string to the symbols and comparing them using the total order implied by string<?. It is not a requirement that the hash function of symbol-comparator be consistent with the hash function of string-comparator, however.

exact-integer-comparator

integer-comparator

rational-comparator

real-comparator

complex-comparator

number-comparator

These comparators compare exact integers, integers, rational numbers, real numbers, complex numbers, and any numbers using the total order implied by <. They must be compatible with the R5RS numerical tower in the following sense: If S is a subtype of the numerical type T and the two objects are members of S , then the equality predicate and comparison procedures (but not necessarily the hash function) of S-comparator and T-comparator compute the same results on those objects.

Since non-real numbers cannot be compared with <, the following least-surprising ordering is defined: If the real parts are < or >, so are the numbers; otherwise, the numbers are ordered by their imaginary parts. This can still produce surprising results if one real part is exact and the other is inexact.

pair-comparator

This comparator compares pairs using default-comparator (see below) on their cars. If the cars are not equal, that value is returned. If they are equal, default-comparator is used on their cdrs and that value is returned.

list-comparator

This comparator compares lists lexicographically, as follows:

- The empty list compares equal to itself.
- The empty list compares less than any non-empty list.
- Two non-empty lists are compared by comparing their cars. If the cars are not equal when compared using default-comparator (see below), then the result is the result of that comparison. Otherwise, the cdrs are compared using list-comparator.

vector-comparator

`bytevector-comparator`

These comparators compare vectors and bytevectors by comparing their lengths. A shorter argument is always less than a longer one. If the lengths are equal, then each element is compared in turn using default-comparator (see below) until a pair of unequal elements is found, in which case the result is the result of that comparison. If all elements are equal, the arguments are equal.

If the implementation does not support bytevectors, bytevector-comparator has a type testing procedure that always returns #f.

The default comparator

`default-comparator`

This is a comparator that accepts any two Scheme values (with the exceptions listed in the Limitations section) and orders them in some implementation-defined way, subject to the following conditions:

- The following ordering between types must hold: the empty list precedes pairs, which precede booleans, which precede characters, which precede strings, which precede symbols, which precede numbers, which precede vectors, which precede bytevectors, which precede all other objects.
- When applied to pairs, booleans, characters, strings, symbols, numbers, vectors, or bytevectors, its behavior must be the same as pair-comparator, boolean-comparator, character-comparator, string-comparator, symbol-comparator, number-comparator, vector-comparator, and bytevector-comparator respectively. The same should be true when applied to an object or objects of a type for which a standard comparator is defined elsewhere.
- Given disjoint types a and b, one of three conditions must hold:
- All objects of type a compare less than all objects of type b.
- All objects of type a compare greater than all objects of type b.
- All objects of either type a or type b compare equal to each other. This is not permitted for any of the standard types mentioned above.

Comparator constructors

`(make-comparator type-test equality compare hash)`

Returns a comparator which bundles the type-test, equality, compare, and hash procedures provided. As a convenience, the following additional values are accepted:

- If type-test is #t, a type-test procedure that accepts any arguments is provided.
- If equality is #t, an equality predicate is provided that returns #t iff compare returns 0.
- If compare or hash is #f, a procedure is provided that signals an error on application. The predicates comparator-comparison-procedure? and/or comparator-hash-function?, respectively, will return #f in these cases.

`(make-inexact-real-comparator epsilon rounding nan-handling)`

Returns a comparator that compares inexact real numbers including NaNs as follows: if after rounding to the nearest epsilon they are the same, they compare equal; otherwise they compare as specified by <. The direction of rounding is specified by the rounding argument, which is either a procedure accepting two arguments (the number and epsilon, or else one of the symbols floor, ceiling, truncate, or round.

The argument nan-handling specifies how to compare NaN arguments to non-NaN arguments. If it is a procedure, the procedure is invoked on the other argument if either argument is a NaN. If it is the symbol min, NaN values precede all other values; if it is the symbol max, they follow all other values, and if it is the symbol error, an error is signaled if a NaN value is compared. If both arguments are NaNs, however, they always compare as equal.

`(make-list-comparator element-comparator)`

`(make-vector-comparator element-comparator)`

`(make-bytevector-comparator element-comparator)`

These procedures return comparators which compare two lists, vectors, or bytevectors in the same way as list-comparator, vector-comparator, and bytevector-comparator respectively, but using element-comparator rather than default-comparator.

If the implementation does not support bytevectors, the result of invoking make-bytevector-comparator is a comparator whose type testing procedure always returns #f.

`(make-listwise-comparator type-test element-comparator empty? head tail)`

Returns a comparator which compares two objects that satisfy type-test as if they were lists, using the empty? procedure to determine if an object is empty, and the head and tail procedures to access particular elements.

`(make-vectorwise-comparator type-test element-comparator length ref)`

Returns a comparator which compares two objects that satisfy type-test as if they were vectors, using the length procedure to determine the length of the object, and the ref procedure to access a particular element.

`(make-car-comparator comparator)`

Returns a comparator that compares pairs on their cars alone using comparator.

`(make-cdr-comparator comparator)`

Returns a comparator that compares pairs on their cdrs alone using comparator.

`(make-pair-comparator car-comparator cdr-comparator)`

Returns a comparator that compares pairs first on their cars using car-comparator. If the cars are equal, it compares the cdrs using cdr-comparator.

`(make-improper-list-comparator element-comparator)`

Returns a comparator that compares arbitrary objects as follows: the empty list precedes all pairs, which precede all other objects. Pairs are compared as if with (make-pair-comparator element-comparator element-comparator). All other objects are compared using element-comparator.

`(make-selecting-comparator comparator1 comparator2 ...)`

Returns a comparator whose procedures make use of the comparators as follows:

The type test predicate passes its argument to the type test predicates of comparators in the sequence given. If any of them returns #t, so does the type test predicate; otherwise, it returns #f.

The arguments of the equality, compare, and hash functions are passed to the type test predicate of each comparator in sequence. The first comparator whose type test predicate is satisfied on all the arguments is used when comparing those arguments. All other comparators are ignored. If no type test predicate is satisfied, an error is signaled.

`(make-refining-comparator comparator1 comparator2 ...)`

Returns a comparator that makes use of the comparators in the same way as make-selecting-comparator, except that its procedures can look past the first comparator whose type test predicate is satisfied. If the comparison procedure of that comparator returns zero, then the next comparator whose type test predicate is satisfied is tried in place of it until one returns a non-zero value. If there are no more such comparators, then the comparison procedure returns zero. The equality predicate is defined in the same way. If no type test predicate is satisfied, an error is signaled.

The hash function of the result returns a value which depends, in an implementation-defined way, on the results of invoking the hash functions of the comparators whose type test predicates are satisfied on its argument. In particular, it may depend solely on the first or last such hash function. If no type test predicate is satisfied, an error is signaled.

This procedure is analogous to the expression type refine-compare from SRFI 67.

`(make-reverse-comparator comparator)`

Returns a comparator that behaves like comparator, except that the compare procedure returns 1, 0, and -1 instead of -1, 0, and 1 respectively. This allows ordering in reverse.

`(make-debug-comparator comparator)`

Returns a comparator that behaves exactly like comparator, except that whenever any of its procedures are invoked, it verifies all the programmer responsibilities (except stability), and an error is signaled if any of them are violated. Because it requires three arguments, transitivity is not tested on the first call to a debug comparator; it is tested on all future calls using an arbitrarily chosen argument from the previous invocation. Note that this may cause unexpected storage leaks.

Wrapped equality predicates

`eq-comparator`

`eqv-comparator`

`equal-comparator`

The equality predicates of these comparators are eq?, eqv?, and equal? respectively. When their comparison procedures are applied to non-equal objects, their behavior is implementation-defined. The type test predicates always return #t.

These comparators accept circular structure (in the case of equal-comparator, provided the implementation's equal does so) and NaNs.

Accessors

`(comparator-type-test-procedure comparator)`

Returns the type test predicate of comparator.

`(comparator-equality-predicate comparator)`

Returns the equality predicate of comparator.

`(comparator-comparison-procedure comparator)`

Returns the comparison procedure of comparator.

`(comparator-hash-function comparator)`

Returns the hash function of comparator.

Primitive applicators

`(comparator-test-type comparator obj)`

Invokes the type test predicate of comparator on obj and returns what it returns.

`(comparator-check-type comparator obj)`

Invokes the type test predicate of comparator on obj and returns true if it returns true and signals an error otherwise.

`(comparator-equal? comparator obj1 obj2)`

Invokes the equality predicate of comparator on obj1 and obj2 and returns what it returns.

`(comparator-compare comparator obj1 obj2)`

Invokes the comparison procedure of comparator on obj1 and obj2 and returns what it returns.

`(comparator-hash comparator obj)`

Invokes the hash function of comparator on obj and returns what it returns.

Comparison procedure constructors

`(make-comparison< lt-pred)`

`(make-comparison> gt-pred)`

`(make-comparison<= le-pred)`

`(make-comparison>= ge-pred)`

`(make-comparison=/< eq-pred lt-pred)`

`(make-comparison=/> eq-pred gt-pred)`

These procedures return a comparison procedure, given a less-than predicate, a greater-than predicate, a less-than-or-equal-to predicate, a greater-than-or-equal-to predicate, or the combination of an equality predicate and either a less-than or a greater-than predicate.

Comparison syntax

The following expression types allow the convenient use of comparison procedures.

`(if3 <expr> <less> <equal> <greater>)`

The expression `<expr>` is evaluated; it will typically, but not necessarily, be a call on a comparison procedure. If the result is -1, `<less>` is evaluated and its value(s) are returned; if the result is 0, `<equal>` is evaluated and its value(s) are returned; if the result is 1, `<greater>` is evaluated and its value(s) are returned. Otherwise an error is signaled.

`(if=? <expr> <consequent> [ <alternate> ])`

`(if<? <expr> <consequent> [ <alternate> ])`

`(if>? <expr> <consequent> [ <alternate> ])`

`(if<=? <expr> <consequent> [ <alternate> ])`

(if>=? <expr> <consequent> [<alternate>])

(if-not=? <expr> <consequent> [<alternate>])

The expression `<expr>` is evaluated; it will typically, but not necessarily, be a call on a comparison procedure. It is an error if its value is not -1, 0, or 1. If the value is consistent with the specified relation, `<consequent>` is evaluated and its value(s) are returned. Otherwise, if `<alternate>` is present, it is evaluated and its value(s) are returned; if it is absent, an unspecified value is returned.

Comparison predicates

(=? comparator object1 object2 object3 ...)

(<? comparator object1 object2 object3 ...)

(>? comparator object1 object2 object3 ...)

(<=? comparator object1 object2 object3 ...)

(>=? comparator object1 object2 object3 ...)

These procedures are analogous to the number, character, and string comparison predicates of Scheme. They allow the convenient use of comparators in situations where the expression types are not usable. They are also analogous to the similarly named procedures SRFI 67, but handle arbitrary numbers of arguments, which in SRFI 67 requires the use of the variants whose names begin with chain.

These procedures apply the comparison procedure of comparator to the objects as follows. If the specified relation returns #t for all objecti and objectj where n is the number of objects and 1 <= i < j <= n, then the procedures return #t, but otherwise #f.

The order in which the values are compared is unspecified. Because the relations are transitive, it suffices to compare each object with its successor.

Comparison predicate constructors

(make=? comparator)

(make<? comparator)

`(make>? comparator)`

`(make<=? comparator)`

`(make>=? comparator)`

These procedures return predicates which, when applied to two or more arguments, return #t if comparing obj1 and obj2 using the equality or comparison procedures of comparator shows that the objects bear the specified relation to one another. Such predicates can be used in contexts that do not understand or expect comparators.

Interval (ternary) comparison predicates

These procedures return true or false depending on whether an object is contained in an open, closed, or half-open interval. All comparisons are done in the sense of comparator, which is default-comparator if omitted.

`(in-open-interval? [comparator] obj1 obj2 obj3)`

Return #t if obj1 is less than obj2, which is less thanobj3, and #f otherwise.

`(in-closed-interval? [comparator] obj1 obj2 obj3)`

Returns #t if obj1 is less than or equal to obj2, which is less than or equal to obj3, and #f otherwise.

`(in-open-closed-interval? [comparator] obj1 obj2 obj3)`

Returns #t if obj1 is less than obj2, which is less than or equal to obj3, and #f otherwise.

`(in-closed-open-interval? [comparator] obj1 obj2 obj3)`

Returns #t if obj1 is less than or equal to obj2, which is less than obj3, and #f otherwise.

Min/max comparison procedures

`(comparator-min comparator object1 object2 ...)`

`(comparator-max comparator object1 object2 ...)`

These procedures are analogous to min and max respectively. They apply the comparison procedure of comparator to the objects to find and return a minimal (or maximal) object. The order in which the values are compared is unspecified.

`(srfi srfi-132)`

This is based on SRFI-132.

Abstract

TODO

Reference

TODO

`(srfi srfi-133)`

This library is based on SRFI-133.

Abstract

TODO

Reference

TODO

`(srfi srfi-134)`

This is based on SRFI-134.

Abstract

TODO

Reference

TODO

`(srfi srfi-135)`

This is based on SRFI-135.

Abstract

TODO

Reference

TODO

`(srfi srfi-141)`

This is based on SRFI-141.

Abstract

TODO

Reference

TODO

`(srfi srfi-143)`

This is based on SRFI-143.

Abstract

TODO

Reference

TODO

`(srfi srfi-144)`

This is based on SRFI-144.

Abstract

TODO

Reference

TODO

`(srfi srfi-145)`

This is based on SRFI-145.

Abstract

TODO

Reference

`(assume obj)` syntax

`(srfi srfi-146)`

This library is based on SRFI-146.

Abstract

Mappings are finite sets of associations, where each association is a pair consisting of a key and an arbitrary Scheme value. The keys are elements of a suitable domain. Each mapping holds no more than one association with the same key. The fundamental mapping operation is retrieving the value of an association stored in the mapping when the key is given.

Reference

mapping

Construtors

`(mapping comparator [key value] ...)`

Returns a newly allocated mapping. The comparator argument is used to control and distinguish the keys of the mapping. The args alternate between keys and values and are used to initialize the mapping. In particular, the number of args has to be even. Earlier associations with equal keys take precedence over later arguments.

`(mapping-unfold stop? mapper successor seed comparator)`

Create a newly allocated mapping as if by mapping using comparator. If the result of applying the predicate stop? to seed is true, return the mapping. Otherwise, apply the procedure mapper to seed. Mapper returns two values which are added to the mapping as the key and the value, respectively. Then get a new seed by applying the procedure successor to seed, and repeat this algorithm. Associations earlier in the list take precedence over those that come later.

`(mapping/ordered)`

`(mapping-unfold/ordered`

These are the same as mapping and mapping-unfold, except that it is an error if the keys are not in order, and they may be more efficient.

Predicates

`(mapping? obj)`

Returns #t if obj is a mapping, and #f otherwise.

`(mapping-contains? mapping key)`

Returns #t if key is the key of an association of mapping and #f otherwise.

`(mapping-empty? mapping)`

Returns #t if mapping has no associations and #f otherwise.

`(mapping-disjoint? mapping1 mapping2)`

Returns #t if mapping1 and mapping2 have no keys in common and #f otherwise.

Accessors

The following three procedures, given a key, return the corresponding value.

`(mapping-ref mapping key [failure [success]])`

Extracts the value associated to key in the mapping mapping, invokes the procedure success in tail context on it, and returns its result; if success is not provided, then the value itself is returned. If key is not contained in mapping and failure is supplied, then failure is invoked in tail context on no arguments and its values are returned. Otherwise, it is an error.

`(mapping-ref/default mapping key default)`

`(mapping-key-comparator mapping)`

Returns the comparator used to compare the keys of the mapping mapping.

Updaters

`(mapping-adjoin mapping arg ...)`

The mapping-adjoin procedure returns a newly allocated mapping that uses the same comparator as the mapping mapping and contains all the associations of mapping, and in addition new associations by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, the previous association prevails and the new association is skipped. It is an error to add an association to mapping whose key that does not return #t when passed to the type test procedure of the comparator.

`(mapping-adjoin! mapping arg ...)`

The mapping-adjoin! procedure is the same as mapping-adjoin, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-set mapping arg ...)`

The mapping-set procedure returns a newly allocated mapping that uses the same comparator as the mapping mapping and contains all the associations of mapping, and in addition new associations by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, it is deleted. It is an error to add an association to mapping whose key that does not return #t when passed to the type test procedure of the comparator.

`(mapping-set! mapping arg ...)`

The mapping-set! procedure is the same as mapping-set, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-replace mapping key value)`

The mapping-replace procedure returns a newly allocated mapping that uses the same comparator as the mapping mapping and contains all the associations of mapping except as follows: If key is equal (in the sense of mapping's comparator) to an existing key of mapping, then the association for that key is omitted and replaced the association defined by the pair key and value. If there is no such key in mapping, then mapping is returned unchanged.

`(mapping-replace! mapping key value)`

The mapping-replace! procedure is the same as mapping-replace, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-delete mapping key ...)`

`(mapping-delete! mapping key ...)`

`(mapping-delete-all mapping key-list)`

`(mapping-delete-all! mapping key-list)`

The mapping-delete procedure returns a newly allocated mapping containing all the associations of the mapping mapping except for any whose keys are equal (in the sense of mapping's comparator) to one or more of the keys. Any key that is not equal to some key of the mapping is ignored.

The mapping-delete! procedure is the same as mapping-delete, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

The mapping-delete-all and mapping-delete-all! procedures are the same as mapping-delete and mapping-delete!, respectively, except that they accept a single argument which is a list of keys whose associations are to be deleted.

`(mapping-intern mapping key failure)`

Extracts the value associated to key in the mapping mapping, and returns mapping and the value as two values. If key is not contained in mapping, failure is invoked on no arguments. The procedure then returns two values, a newly allocated mapping that uses the same comparator as the mapping and contains all the associations of mapping, and in addition a new association mapping key to the result of invoking failure, and the result of invoking failure.

`(mapping-intern! mapping key failure)`

The mapping-intern! procedure is the same as mapping-intern, except that it is permitted to mutate and return the mapping argument as its first value rather than allocating a new mapping.

`(mapping-update mapping key updater [failure [success]])`

TODO

`(mapping-update! mapping key updater [failure [success]])`

The mapping-update! procedure is the same as mapping-update, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-update/default mapping key updater default)`

TODO

`(mapping-update!/default mapping key updater default)`

The mapping-update!/default procedure is the same as mapping-update/default, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-pop mapping [failure])`

The mapping-pop procedure exported from (srfi 146) chooses the association with the least key from mapping and returns three values, a newly allocated mapping that uses the same comparator as mapping and contains all associations of mapping except the chosen one, and the key and the value of the chosen association. If mapping contains no association and failure is supplied, then failure is invoked in tail context on no arguments and its values returned. Otherwise, it is an error.

`(mapping-pop! mapping [failure])`

The mapping-pop! procedure is the same as mapping-pop, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-search mapping key failure success)`

The mapping mapping is searched in order (that is in the order of the stored keys) for an association with key key. If it is not found, then the failure procedure is tail-called with two continuation arguments, insert and ignore, and is expected to tail-call one of them. If an association with key key is found, then the success procedure is tail-called with the matching key of mapping, the associated value, and two continuations, update and remove, and is expected to tail-call one of them.

It is an error if the continuation arguments are invoked, but not in tail position in the failure and success procedures. It is also an error if the failure and success procedures return to their implicit continuation without invoking one of their continuation arguments.

The effects of the continuations are as follows (where obj is any Scheme object):

- Invoking (insert value obj) causes a mapping to be newly allocated that uses the same comparator as the mapping mapping and contains all the associations of mapping, and in addition a new association mapping key to value.
- Invoking (ignore obj) has no effects; in particular, no new mapping is allocated (but see below).
- Invoking (update new-key new-value obj) causes a mapping to be newly allocated that uses the same comparator as the mapping and contains all

the associations of mapping, except for the association with key key, which is replaced by a new association mapping new-key to new-value.

- Invoking (remove obj) causes a mapping to be newly allocated that uses the same comparator as the mapping and contains all the associations of mapping, except for the association with key key.

In all cases, two values are returned: the possibly newly allocated mapping and obj.

`(mapping-search! mapping key failure success)`

The mapping-search! procedure is the same as mapping-search, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

The whole mapping

`(mapping-size mapping)`

Returns the number of associations in mapping as an exact integer.

`(mapping-find predicate mapping failure)`

Returns the association with the least key of the mapping mapping consisting of a key and value as two values such that predicate returns a true value when invoked with key and value as arguments, or the result of tail-calling failure with no arguments if there is none. There are no guarantees how many times and with which keys and values predicate is invoked.

`(mapping-count predicate mapping)`

Returns the number of associations of the mapping mapping that satisfy predicate (in the sense of mapping-find) as an exact integer. There are no guarantees how many times and with which keys and values predicate is invoked.

`(mapping-any? predicate mapping)`

Returns #t if any association of the mapping mapping satisfies predicate (in the sense of mapping-find), or #f otherwise. There are no guarantees how many times and with which keys and values predicate is invoked.

`(mapping-every? predicate mapping)`

Returns #t if every association of the mapping mapping satisfies predicate (in the sense of mapping-find), or #f otherwise. There are no guarantees how many times and with which keys and values predicate is invoked.

`(mapping-keys mapping)`

Returns a newly allocated list of all the keys in increasing order in the mapping mapping.

`(mapping-values mapping)`

Returns a newly allocated list of all the values in increasing order of the keys in the mapping mapping.

`(mapping-entries mapping)`

Returns two values, a newly allocated list of all the keys in the mapping mapping, and a newly allocated list of all the values in the mapping mapping in increasing order of the keys.

Mapping and folding

`(mapping-map proc comparator mapping)`

Applies proc, which returns two values, on two arguments, the key and value of each association of mapping in increasing order of the keys and returns a newly allocated mapping that uses the comparator comparator, and which contains the results of the applications inserted as keys and values.

`(mapping-map->list proc mapping)`

Calls proc for every association in increasing order of the keys in the mapping mapping with two arguments: the key of the association and the value of the association. The values returned by the invocations of proc are accumulated into a list, which is returned.

`(mapping-for-each proc mapping)`

Invokes proc for every association in the mapping mapping in increasing order of the keys, discarding the returned values, with two arguments: the key of the association and the value of the association. Returns an unspecified value.

`(mapping-fold proc nil mapping)`

Invokes proc for each association of the mapping mapping in increasing order of the keys with three arguments: the key of the association, the value of the association, and an accumulated result of the previous invocation. For the first invocation, nil is used as the third argument. Returns the result of the last invocation, or nil if there was no invocation.

`(mapping-filter predicate mapping)`

Returns a newly allocated mapping with the same comparator as the mapping mapping, containing just the associations of mapping that satisfy predicate (in the sense of mapping-find).

`(mapping-filter! predicate mapping)`

A linear update procedure that returns a mapping containing just the associations of mapping that satisfy predicate.

`(mapping-remove predicate mapping)`

Returns a newly allocated mapping with the same comparator as the mapping mapping, containing just the associations of mapping that do not satisfy predicate (in the sense of mapping-find).

`(mapping-remove! predicate mapping)`

A linear update procedure that returns a mapping containing just the associations of mapping that do not satisfy predicate.

`(mapping-partition predicate mapping)`

Returns two values: a newly allocated mapping with the same comparator as the mapping mapping that contains just the associations of mapping that satisfy predicate (in the sense of mapping-find), and another newly allocated mapping, also with the same comparator, that contains just the associations of mapping that do not satisfy predicate.

`(mapping-partition! predicate mapping)`

A linear update procedure that returns two mappings containing the associations of mapping that do and do not, respectively, satisfy predicate.

Copying and conversion

`(mapping-copy mapping)`

Returns a newly allocated mapping containing the associations of the mapping mapping, and using the same comparator.

`(mapping->alist mapping)`

Returns a newly allocated association list containing the associations of the mapping in increasing order of the keys. Each association in the list is a pair whose car is the key and whose cdr is the associated value.

`(alist->mapping comparator alist)`

Returns a newly allocated mapping, created as if by mapping using the comparator comparator, that contains the associations in the list, which consist of a pair whose car is the key and whose cdr is the value. Associations earlier in the list take precedence over those that come later.

`(alist->mapping! mapping alist)`

A linear update procedure that returns a mapping that contains the associations of both mapping and alist. Associations in the mapping and those earlier in the list take precedence over those that come later.

Submappings

All predicates in this subsection take a comparator argument, which is a comparator used to compare the values of the associations stored in the mappings.

Associations in the mappings are equal if their keys are equal with mappings' key comparator and their values are equal with the given comparator. Two mappings are equal if and only if their associations are equal, respectively.

Note: None of these five predicates produces a total order on mappings. In particular, mapping=?, mapping<?, and mapping>? do not obey the trichotomy law.

`(mapping=? comparator mapping1 mapping2 ...)`

Returns #t if each mapping mapping contains the same associations, and #f otherwise.

Furthermore, it is explicitly not an error if mapping=? is invoked on mappings that do not share the same (key) comparator. In that case, #f is returned.

`(mapping<? comparator mapping1 mapping2 ...)`

Returns #t if the set of associations of each mapping mapping other than the last is a proper subset of the following mapping, and #f otherwise.

`(mapping>? comparator mapping1 mapping2 ...)`

Returns #t if each mapping mapping contains the same associations, and #f otherwise.

Furthermore, it is explicitly not an error if mapping=? is invoked on mappings that do not share the same (key) comparator. In that case, #f is returned.

`(mapping<=? comparator mapping1 mapping2 ...)`

Returns #t if the set of associations of each mapping mapping other than the last is a subset of the following mapping, and #f otherwise.

`(mapping>=? comparator mapping1 mapping2 ...)`

Returns #t if the set of associations of each mapping mapping other than the last is a superset of the following mapping, and #f otherwise.

Set theory operations

`(mapping-union mapping1 mapping2 ...)`

`(mapping-intersection mapping1 mapping2 ...)`

`(mapping-difference mapping1 mapping2 ...)`

`(mapping-xor mapping1 mapping2 ...)`

Return a newly allocated mapping whose set of associations is the union, intersection, asymmetric difference, or symmetric difference of the sets of associations of the mappings mappings. Asymmetric difference is extended to more than two mappings by taking the difference between the first mapping and the union of the others. Symmetric difference is not extended beyond two mappings. When comparing associations, only the keys are compared. In case of duplicate keys (in the sense of the mappings comparators), associations in the result mapping are drawn from the first mapping in which they appear.

`(mapping-union! mapping1 mapping2 ...)`

`(mapping-intersection! mapping1 mapping2 ...)`

`(mapping-difference! mapping1 mapping2 ...)`

`(mapping-xor! mapping1 mapping2 ...)`

These procedures are the linear update analogs of the corresponding pure functional procedures above.

Additional procedures

`(mapping-min-key mapping)`

`(mapping-max-key mapping)`

Returns the least/greatest key contained in the mapping mapping. It is an error for mapping to be empty.

`(mapping-min-value mapping)`

`(mapping-max-value mapping)`

Returns the value associated with the least/greatest key contained in the mapping mapping. It is an error for mapping to be empty.

`(mapping-min-entry mapping)`

`(mapping-max-entry mapping)`

Returns the entry associated with the least/greatest key contained in the mapping mapping as two values, the key and its associated value. It is an error for mapping to be empty.

`(mapping-key-predecessor mapping obj failure)`

`(mapping-key-successor mapping obj failure)`

Returns the key contained in the mapping mapping that immediately precedes/succeeds obj in the mapping's order of keys. If no such key is contained in mapping (because obj is the minimum/maximum key, or because mapping is empty), returns the result of tail-calling the thunk failure.

`(mapping-range= mapping obj)`

`(mapping-range< mapping obj)`

`(mapping-range> mapping obj)`

`(mapping-range<= mapping obj)`

`(mapping-range>= mapping obj)`

Returns a mapping containing only the associations of the mapping whose keys are equal to, less than, greater than, less than or equal to, or greater than or equal to obj.

`(mapping-range=! mapping obj)`

`(mapping-range<! mapping obj)`

`(mapping-range>! mapping obj)`

`(mapping-range<=! mapping obj)`

`(mapping-range>=! mapping obj)`

Linear update procedures returning a mapping containing only the associations of the mapping whose keys are equal to, less than, greater than, less than or equal to, or greater than or equal to obj.

`(mapping-split mapping obj)`

Returns five values, equivalent to the results of invoking (mapping-range< mapping obj), (mapping-range<= mapping obj), (mapping-range= mapping obj), (mapping-range>= mapping obj), and (mapping-range> mapping obj), but may be more efficient.

`(mapping-split! mapping obj)`

The mapping-split! procedure is the same as mapping-split, except that it is permitted to mutate and return the mapping rather than allocating a new mapping.

`(mapping-catenate comparator mapping1 key value mapping2)`

Returns a newly allocated mapping using the comparator comparator whose set of associations is the union of the sets of associations of the mapping mapping1, the association mapping key to value, and the associations of mapping2. It is an error if the keys contained in mapping1 in their natural order, the key key, and the keys contained in mapping2 in their natural order (in that order) do not form a strictly monotone sequence with respect to the ordering of comparator.

`(mapping-catenate! comparator mapping1 key value mapping2)`

Returns a newly allocated mapping using the comparator comparator whose set of associations is the union of the sets of associations of the mapping mapping1, the association mapping key to value, and the associations of mapping2. It is an error if the keys contained in mapping1 in their natural order, the key key, and the keys contained in mapping2 in their natural order (in that order) do not form a strictly monotone sequence with respect to the ordering of comparator.

`(mapping-map/monotone proc comparator mapping)`

Equivalent to (mapping-map proc comparator mapping), but it is an error if proc does not induce a strictly monotone mapping between the keys with respect to the ordering of the comparator of mapping and the ordering of comparator. Maybe be implemented more efficiently than mapping-map.

`(mapping-map/monotone! proc comparator mapping)`

The mapping-map/monotone! procedure is the same as mapping-map/monotone, except that it is permitted to mutate and return the mapping argument rather than allocating a new mapping.

`(mapping-fold/reverse proc nil mapping)`

Equivalent to (mapping-fold proc nil mapping) except that the associations are processed in reverse order with respect to the natural ordering of the keys.

Comparators

`(comparator? obj)`

Type predicate for comparators as exported by `(scheme comparator)`.

`mapping-comparator`

mapping-comparator is constructed by invoking make-mapping-comparator on (make-default-comparator).

`(make-mapping-comparator comparator)`

Returns a comparator for mappings that is compatible with the equality predicate (mapping=? comparator mapping1 mapping2). If make-mapping-comparator is imported from (srfi 146), it provides a (partial) ordering predicate that is applicable to pairs of mappings with the same (key) comparator. If (make-hashmap-comparator) is imported from (srfi 146 hash), it provides an implementation-dependent hash function.

If make-mapping-comparator is imported from (srfi 146), the lexicographic ordering with respect to the keys (and, in case a tiebreak is necessary, with respect to the ordering of the values) is used for mappings sharing a comparator.

The existence of comparators returned by make-mapping-comparator allows mappings whose keys are mappings themselves, and it allows to compare mappings whose values are mappings.

hash mapping

Construtors

(hashmap comparator [key value] ...)

Returns a newly allocated hashmap. The comparator argument is used to control and distinguish the keys of the hashmap. The args alternate between keys and values and are used to initialize the hashmap. In particular, the number of args has to be even. Earlier associations with equal keys take precedence over later arguments.

(hashmap-unfold stop? mapper successor seed comparator)

Create a newly allocated hashmap as if by hashmap using comparator. If the result of applying the predicate stop? to seed is true, return the hashmap. Otherwise, apply the procedure mapper to seed. Mapper returns two values which are added to the hashmap as the key and the value, respectively. Then get a new seed by applying the procedure successor to seed, and repeat this algorithm. Associations earlier in the list take precedence over those that come later.

Predicates

(hashmap? obj)

Returns #t if obj is a hashmap, and #f otherwise.

(hashmap-contains? hashmap key)

Returns #t if key is the key of an association of hashmap and #f otherwise.

(hashmap-empty? hashmap)

Returns #t if hashmap has no associations and #f otherwise.

(hashmap-disjoint? hashmap1 hashmap2)

Returns #t if hashmap1 and hashmap2 have no keys in common and #f otherwise.

Accessors

The following three procedures, given a key, return the corresponding value.

(hashmap-ref hashmap key [failure [success]])

Extracts the value associated to key in the hashmap hashmap, invokes the procedure success in tail context on it, and returns its result; if success is not provided, then the value itself is returned. If key is not contained in hashmap and failure is supplied, then failure is invoked in tail context on no arguments and its values are returned. Otherwise, it is an error.

(hashmap-ref/default hashmap key default)

(hashmap-key-comparator hashmap)

Returns the comparator used to compare the keys of the hashmap hashmap.

Updaters

(hashmap-adjoin hashmap arg ...)

The hashmap-adjoin procedure returns a newly allocated hashmap that uses the same comparator as the hashmap hashmap and contains all the associations of hashmap, and in addition new associations by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, the previous association prevails and the new association is skipped. It is an error to add an association to hashmap whose key that does not return #t when passed to the type test procedure of the comparator.

(hashmap-adjoin! hashmap arg ...)

The hashmap-adjoin! procedure is the same as hashmap-adjoin, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

(hashmap-set hashmap arg ...)

The hashmap-set procedure returns a newly allocated hashmap that uses the same comparator as the hashmap hashmap and contains all the associations of hashmap, and in addition new associations by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, it is deleted. It is an error to add an association to hashmap whose key that does not return #t when passed to the type test procedure of the comparator.

(hashmap-set! hashmap arg ...)

The hashmap-set! procedure is the same as hashmap-set, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

(hashmap-replace hashmap key value)

The hashmap-replace procedure returns a newly allocated hashmap that uses the same comparator as the hashmap hashmap and contains all the associations of hashmap except as follows: If key is equal (in the sense of hashmap's comparator) to an existing key of hashmap, then the association for that key is omitted and replaced the association defined by the pair key and value. If there is no such key in hashmap, then hashmap is returned unchanged.

(hashmap-replace! hashmap key value)

The hashmap-replace! procedure is the same as hashmap-replace, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

(hashmap-delete hashmap key ...)

(hashmap-delete! hashmap key ...)

(hashmap-delete-all hashmap key-list)

(hashmap-delete-all! hashmap key-list)

The hashmap-delete procedure returns a newly allocated hashmap containing all the associations of the hashmap hashmap except for any whose keys are equal (in the sense of hashmap's comparator) to one or more of the keys. Any key that is not equal to some key of the hashmap is ignored.

The hashmap-delete! procedure is the same as hashmap-delete, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

The hashmap-delete-all and hashmap-delete-all! procedures are the same as hashmap-delete and hashmap-delete!, respectively, except that they accept a single argument which is a list of keys whose associations are to be deleted.

`(hashmap-intern hashmap key failure)`

Extracts the value associated to key in the hashmap hashmap, and returns hashmap and the value as two values. If key is not contained in hashmap, failure is invoked on no arguments. The procedure then returns two values, a newly allocated hashmap that uses the same comparator as the hashmap and contains all the associations of hashmap, and in addition a new association hashmap key to the result of invoking failure, and the result of invoking failure.

`(hashmap-intern! hashmap key failure)`

The hashmap-intern! procedure is the same as hashmap-intern, except that it is permitted to mutate and return the hashmap argument as its first value rather than allocating a new hashmap.

`(hashmap-update hashmap key updater [failure [success]])`

TODO

`(hashmap-update! hashmap key updater [failure [success]])`

The hashmap-update! procedure is the same as hashmap-update, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

`(hashmap-update/default hashmap key updater default)`

TODO

`(hashmap-update!/default hashmap key updater default)`

The hashmap-update!/default procedure is the same as hashmap-update/default, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

`(hashmap-pop hashmap [failure])`

The hashmap-pop procedure exported from (srfi 146) chooses the association with the least key from hashmap and returns three values, a newly allocated hashmap that uses the same comparator as hashmap and contains all associations of hashmap except the chosen one, and the key and the value of the chosen association. If hashmap contains no association and failure is supplied, then failure is invoked in tail context on no arguments and its values returned. Otherwise, it is an error.

(hashmap-pop! hashmap [failure])

The hashmap-pop! procedure is the same as hashmap-pop, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

(hashmap-search hashmap key failure success)

The hashmap hashmap is searched in order (that is in the order of the stored keys) for an association with key key. If it is not found, then the failure procedure is tail-called with two continuation arguments, insert and ignore, and is expected to tail-call one of them. If an association with key key is found, then the success procedure is tail-called with the matching key of hashmap, the associated value, and two continuations, update and remove, and is expected to tail-call one of them.

It is an error if the continuation arguments are invoked, but not in tail position in the failure and success procedures. It is also an error if the failure and success procedures return to their implicit continuation without invoking one of their continuation arguments.

The effects of the continuations are as follows (where obj is any Scheme object):

- Invoking (insert value obj) causes a hashmap to be newly allocated that uses the same comparator as the hashmap hashmap and contains all the associations of hashmap, and in addition a new association hashmap key to value.
- Invoking (ignore obj) has no effects; in particular, no new hashmap is allocated (but see below).
- Invoking (update new-key new-value obj) causes a hashmap to be newly allocated that uses the same comparator as the hashmap and contains all the associations of hashmap, except for the association with key key, which is replaced by a new association hashmap new-key to new-value.
- Invoking (remove obj) causes a hashmap to be newly allocated that uses the same comparator as the hashmap and contains all the associations of hashmap, except for the association with key key.

In all cases, two values are returned: the possibly newly allocated hashmap and obj.

(hashmap-search! hashmap key failure success)

The hashmap-search! procedure is the same as hashmap-search, except that it is permitted to mutate and return the hashmap argument rather than allocating a new hashmap.

The whole hashmap

`(hashmap-size hashmap)`

Returns the number of associations in hashmap as an exact integer.

`(hashmap-find predicate hashmap failure)`

Returns the association with the least key of the hashmap hashmap consisting of a key and value as two values such that predicate returns a true value when invoked with key and value as arguments, or the result of tail-calling failure with no arguments if there is none. There are no guarantees how many times and with which keys and values predicate is invoked.

`(hashmap-count predicate hashmap)`

Returns the number of associations of the hashmap hashmap that satisfy predicate (in the sense of hashmap-find) as an exact integer. There are no guarantees how many times and with which keys and values predicate is invoked.

`(hashmap-any? predicate hashmap)`

Returns #t if any association of the hashmap hashmap satisfies predicate (in the sense of hashmap-find), or #f otherwise. There are no guarantees how many times and with which keys and values predicate is invoked.

`(hashmap-every? predicate hashmap)`

Returns #t if every association of the hashmap hashmap satisfies predicate (in the sense of hashmap-find), or #f otherwise. There are no guarantees how many times and with which keys and values predicate is invoked.

`(hashmap-keys hashmap)`

Returns a newly allocated list of all the keys in increasing order in the hashmap hashmap.

`(hashmap-values hashmap)`

Returns a newly allocated list of all the values in increasing order of the keys in the hashmap hashmap.

(hashmap-entries hashmap)

Returns two values, a newly allocated list of all the keys in the hashmap hashmap, and a newly allocated list of all the values in the hashmap hashmap in increasing order of the keys.

Hashmap and folding

(hashmap-map proc comparator hashmap)

Applies proc, which returns two values, on two arguments, the key and value of each association of hashmap in increasing order of the keys and returns a newly allocated hashmap that uses the comparator comparator, and which contains the results of the applications inserted as keys and values.

(hashmap-map->list proc hashmap)

Calls proc for every association in increasing order of the keys in the hashmap hashmap with two arguments: the key of the association and the value of the association. The values returned by the invocations of proc are accumulated into a list, which is returned.

(hashmap-for-each proc hashmap)

Invokes proc for every association in the hashmap hashmap in increasing order of the keys, discarding the returned values, with two arguments: the key of the association and the value of the association. Returns an unspecified value.

(hashmap-fold proc nil hashmap)

Invokes proc for each association of the hashmap hashmap in increasing order of the keys with three arguments: the key of the association, the value of the association, and an accumulated result of the previous invocation. For the first invocation, nil is used as the third argument. Returns the result of the last invocation, or nil if there was no invocation.

(hashmap-filter predicate hashmap)

Returns a newly allocated hashmap with the same comparator as the hashmap hashmap, containing just the associations of hashmap that satisfy predicate (in the sense of hashmap-find).

`(hashmap-filter! predicate hashmap)`

A linear update procedure that returns a hashmap containing just the associations of hashmap that satisfy predicate.

`(hashmap-remove predicate hashmap)`

Returns a newly allocated hashmap with the same comparator as the hashmap hashmap, containing just the associations of hashmap that do not satisfy predicate (in the sense of hashmap-find).

`(hashmap-remove! predicate hashmap)`

A linear update procedure that returns a hashmap containing just the associations of hashmap that do not satisfy predicate.

`(hashmap-partition predicate hashmap)`

Returns two values: a newly allocated hashmap with the same comparator as the hashmap hashmap that contains just the associations of hashmap that satisfy predicate (in the sense of hashmap-find), and another newly allocated hashmap, also with the same comparator, that contains just the associations of hashmap that do not satisfy predicate.

`(hashmap-partition! predicate hashmap)`

A linear update procedure that returns two hashmaps containing the associations of hashmap that do and do not, respectively, satisfy predicate.

Copying and conversion

`(hashmap-copy hashmap)`

Returns a newly allocated hashmap containing the associations of the hashmap hashmap, and using the same comparator.

`(hashmap->alist hashmap)`

Returns a newly allocated association list containing the associations of the hashmap in increasing order of the keys. Each association in the list is a pair whose car is the key and whose cdr is the associated value.

(alist->hashmap comparator alist)

Returns a newly allocated hashmap, created as if by hashmap using the comparator comparator, that contains the associations in the list, which consist of a pair whose car is the key and whose cdr is the value. Associations earlier in the list take precedence over those that come later.

(alist->hashmap! hashmap alist)

A linear update procedure that returns a hashmap that contains the associations of both hashmap and alist. Associations in the hashmap and those earlier in the list take precedence over those that come later.

Set theory operations

(hashmap-union hashmap1 hashmap2 ...)

(hashmap-intersection hashmap1 hashmap2 ...)

(hashmap-difference hashmap1 hashmap2 ...)

(hashmap-xor hashmap1 hashmap2 ...)

Return a newly allocated hashmap whose set of associations is the union, intersection, asymmetric difference, or symmetric difference of the sets of associations of the hashmaps hashmaps. Asymmetric difference is extended to more than two hashmaps by taking the difference between the first hashmap and the union of the others. Symmetric difference is not extended beyond two hashmaps. When comparing associations, only the keys are compared. In case of duplicate keys (in the sense of the hashmaps comparators), associations in the result hashmap are drawn from the first hashmap in which they appear.

(hashmap-union! hashmap1 hashmap2 ...)

(hashmap-intersection! hashmap1 hashmap2 ...)

(hashmap-difference! hashmap1 hashmap2 ...)

`(hashmap-xor! hashmap1 hashmap2 ...)`

These procedures are the linear update analogs of the corresponding pure functional procedures above.

Comparators

`(comparator? obj)`

Type predicate for comparators as exported by `(scheme comparator)`.

`hashmap-comparator`

hashmap-comparator is constructed by invoking make-hashmap-comparator on (make-default-comparator).

`(make-hashmap-comparator comparator)`

Returns a comparator for hashmaps that is compatible with the equality predicate (hashmap=? comparator hashmap1 hashmap2). If make-hashmap-comparator is imported from (srfi 146), it provides a (partial) ordering predicate that is applicable to pairs of hashmaps with the same (key) comparator. If (make-hashmap-comparator) is imported from (srfi 146 hash), it provides an implementation-dependent hash function.

If make-hashmap-comparator is imported from (srfi 146), the lexicographic ordering with respect to the keys (and, in case a tiebreak is necessary, with respect to the ordering of the values) is used for hashmaps sharing a comparator.

The existence of comparators returned by make-hashmap-comparator allows hashmaps whose keys are hashmaps themselves, and it allows to compare hashmaps whose values are hashmaps.

`(srfi srfi-151)`

This library is based on SRFI-151.

Abstract

This library offers a coherent and comprehensive set of procedures for performing bitwise logical operations on integers.

Reference

`(bitwise-not i)`

Returns the bitwise complement of i; that is, all 1 bits are changed to 0 bits and all 0 bits to 1 bits.

```
(bitwise-not 10) ;; => -11
(bitwise-not -37) ;; => 36
```

The following ten procedures correspond to the useful set of non-trivial two-argument boolean functions. For each such function, the corresponding bitwise operator maps that function across a pair of bitstrings in a bit-wise fashion. The core idea of this group of functions is this bitwise "lifting" of the set of dyadic boolean functions to bitstring parameters.

`(bitwise-and i ...)`

`(bitwise-ior i ...)`

`(bitwise-xor i ...)`

`(bitwise-eqv i ...)`

These operations are associative. When passed no arguments, the procedures return the identity values -1, 0, 0, and -1 respectively.

The bitwise-eqv procedure produces the complement of the bitwise-xor procedure. When applied to three arguments, it does not produce a 1 bit everywhere that a, b and c all agree. That is, it does not produce

```
(bitwise-ior (bitwise-and a b c)
             (bitwise-and (bitwise-not a)
                          (bitwise-not b)
                          (bitwise-not c)))
```

Rather, it produces (bitwise-eqv a (bitwise-eqv b c)) or the equivalent (bitwise-eqv (bitwise-eqv a b) c).

```
(bitwise-ior 3  10)     =>  11
(bitwise-and 11 26)     =>  10
(bitwise-xor 3 10)      =>   9
(bitwise-eqv 37 12)     => -42
(bitwise-and 37 12)     =>   4
```

`(bitwise-nand i j)`

(bitwise-nor i j)

(bitwise-andc1 i j)

(bitwise-andc2 i j)

(bitwise-orc1 i j)

(bitwise-orc2 i j)

These operations are not associative.

```
(bitwise-nand 11 26) =>  -11
(bitwise-nor  11 26) => -28
(bitwise-andc1 11 26) => 16
(bitwise-andc2 11 26) => 1
(bitwise-orc1 11 26) => -2
(bitwise-orc2 11 26) => -17
```

(arithmetic-shift i count)

Returns the arithmetic left shift when count>0; right shift when count<0.

```
(arithmetic-shift 8 2) => 32
(arithmetic-shift 4 0) => 4
(arithmetic-shift 8 -1) => 4
(arithmetic-shift -100000000000000000000000000000000 -100) => -79
```

(bit-count i)

Returns the population count of 1's (i >= 0) or 0's (i < 0). The result is always non-negative.

Compatibility note: The R6RS analogue bitwise-bit-count applies bitwise-not to the population count before returning it if i is negative.

```
(bit-count 0) =>  0
(bit-count -1) =>  0
(bit-count 7) =>  3
(bit-count  13) =>  3 ;Two's-complement binary: ...0001101
(bit-count -13) =>  2 ;Two's-complement binary: ...1110011
(bit-count  30) =>  4 ;Two's-complement binary: ...0011110
(bit-count -30) =>  4 ;Two's-complement binary: ...1100010
(bit-count (expt 2 100)) =>  1
```

```
(bit-count (- (expt 2 100))) =>  100
(bit-count (- (1+ (expt 2 100)))) =>  1
```

`(integer-length i)`

The number of bits needed to represent i, i.e.

```
(ceiling (/ (log (if (negative? integer)
                     (- integer)
                     (+ 1 integer)))
            (log 2)))
```

The result is always non-negative. For non-negative i, this is the number of bits needed to represent i in an unsigned binary representation. For all i, (+ 1 (integer-length i)) is the number of bits needed to represent i in a signed twos-complement representation.

```
(integer-length  0) => 0
(integer-length  1) => 1
(integer-length -1) => 0
(integer-length  7) => 3
(integer-length -7) => 3
(integer-length  8) => 4
(integer-length -8) => 3
```

`(bitwise-if mask i j)`

Merge the bitstrings i and j, with bitstring mask determining from which string to take each bit. That is, if the kth bit of mask is 1, then the kth bit of the result is the kth bit of i, otherwise the kth bit of j.

```
(bitwise-if 3 1 8) => 9
(bitwise-if 3 8 1) => 0
(bitwise-if 1 1 2) => 3
(bitwise-if #b00111100 #b11110000 #b00001111) => #b00110011
```

`(bit-set? index i)`

Is bit index set in bitstring i (where index is a non-negative exact integer)?

Compatibility note: The R6RS analogue bitwise-bit-set? accepts its arguments in the opposite order.

```
(bit-set? 1 1) =>  false
(bit-set? 0 1) =>  true
(bit-set? 3 10) =>  true
(bit-set? 1000000 -1) =>  true
```

```
(bit-set? 2 6) =>  true
(bit-set? 0 6) =>  false
```

(copy-bit index i boolean)

Returns an integer the same as i except in the indexth bit, which is 1 if boolean is #t and 0 if boolean is #f.

Compatibility note: The R6RS analogue bitwise-copy-bit as originally documented has a completely different interface. (bitwise-copy-bit dest index source) replaces the index'th bit of dest with the index'th bit of source. It is equivalent to (bit-field-replace-same dest source index (+ index 1)). However, an erratum made a silent breaking change to interpret the third argument as 0 for a false bit and 1 for a true bit. Some R6RS implementations applied this erratum but others did not.

```
(copy-bit 0 0 #t) => #b1
(copy-bit 2 0 #t) => #b100
(copy-bit 2 #b1111 #f) => #b1011
```

(bit-swap index1 index2 i)

Returns an integer the same as i except that the index1th bit and the index2th bit have been exchanged.

```
(bit-swap 0 2 4) => #b1
```

(any-bit-set? test-bits i)

(every-bit-set? test-bits i)

Determines if any/all of the bits set in bitstring test-bits are set in bitstring i. I.e., returns (not (zero? (bitwise-and test-bits i))) and (= test-bits (bitwise-and test-bits i))) respectively.

```
(any-bit-set? 3 6) => #t
(any-bit-set? 3 12) => #f
(every-bit-set? 4 6) => #t
(every-bit-set? 7 6) => #f
```

(first-set-bit i)

Return the index of the first (smallest index) 1 bit in bitstring i. Return -1 if i contains no 1 bits (i.e., if i is zero).

```
(first-set-bit 1) => 0
(first-set-bit 2) => 1
(first-set-bit 0) => -1
(first-set-bit 40) => 3
(first-set-bit -28) => 2
(first-set-bit (expt  2 99)) => 99
(first-set-bit (expt -2 99)) => 99
```

‘(bit-field i start end)

Returns the field from i, shifted down to the least-significant position in the result.

```
(bit-field #b1101101010 0 4) => #b1010
(bit-field #b1101101010 3 9) => #b101101
(bit-field #b1101101010 4 9) => #b10110
(bit-field #b1101101010 4 10) => #b110110
(bit-field 6 0 1) => 0
(bit-field 6 1 3) => 3
(bit-field 6 2 999) => 1
(bit-field #x100000000000000000000000000000000 128 129) => 1
```

`(bit-field-any? i start end)`

Returns true if any of the field’s bits are set in bitstring i, and false otherwise.

```
(bit-field-any? #b1001001 1 6) => #t
(bit-field-any? #b1000001 1 6) => #f
```

`(bit-field-every? i start end)`

Returns false if any of the field’s bits are not set in bitstring i, and true otherwise.

```
(bit-field-every? #b1011110 1 5) => #t
(bit-field-every? #b1011010 1 5) => #f
```

`(bit-field-clear i start end)`

`(bit-field-set i start end)`

Returns i with the field’s bits set to all 0s/1s.

```
(bit-field-clear #b101010 1 4) => #b100000
(bit-field-set #b101010 1 4) => #b101110
```

(bit-field-replace dest source start end)

Returns dest with the field replaced by the least-significant end-start bits in source.

```
(bit-field-replace #b101010 #b010 1 4) => #b100100
(bit-field-replace #b110 1 0 1) => #b111
(bit-field-replace #b110 1 1 2) => #b110
```

(bit-field-replace-same dest source start end)

Returns dest with its field replaced by the corresponding field in source.

```
(bit-field-replace-same #b1111 #b0000 1 3) => #b1001
```

(bit-field-rotate i count start end)

Returns i with the field cyclically permuted by count bits towards high-order.

Compatibility note: The R6RS analogue bitwise-rotate-bit-field uses the argument ordering i start end count.

```
(bit-field-rotate #b110 0 0 10) => #b110
(bit-field-rotate #b110 0 0 256) => #b110
(bit-field-rotate #x100000000000000000000000000000000 1 0 129) => 1
(bit-field-rotate #b110 1 1 2) => #b110
(bit-field-rotate #b110 1 2 4) => #b1010
(bit-field-rotate #b0111 -1 1 4) => #b1011
```

(bit-field-reverse i start end)

Returns i with the order of the bits in the field reversed.

```
(bit-field-reverse 6 1 3) => 6
(bit-field-reverse 6 1 4) => 12
(bit-field-reverse 1 0 32) => #x80000000
(bit-field-reverse 1 0 31) => #x40000000
(bit-field-reverse 1 0 30) => #x20000000
(bit-field-reverse #x140000000000000000000000000000000 0 129) => 5
```

(bits->list i [len])

(bits->vector i [len])

Returns a list/vector of len booleans corresponding to each bit of the non-negative integer i, returning bit #0 as the first element, bit #1 as the second, and so on. #t is returned for each 1; #f for 0.

```
(bits->list #b1110101)) => (#t #f #t #f #t #t #t)
(bits->list 3 5)) => (#t #t #f #f #f)
(bits->list 6 4)) => (#f #t #t #f)

(bits->vector #b1110101)) => #(#t #f #t #f #t #t #t)
```

(list->bits list)

(vector->bits vector)

Returns an integer formed from the booleans in list/vector, using the first element as bit #0, the second element as bit #1, and so on. It is an error if list/vector contains non-booleans. A 1 bit is coded for each #t; a 0 bit for #f. Note that the result is never a negative integer.

```
(list->bits '(#t #f #t #f #t #t #t)) => #b1110101
(list->bits '(#f #f #t #f #t #f #t #t #t)) => #b111010100
(list->bits '(#f #t #t)) => 6
(list->bits '(#f #t #t #f)) => 6
(list->bits '(#f #f #t #t)) => 12

(vector->bits '#(#t #f #t #f #t #t #t)) => #b1110101
(vector->bits '#(#f #f #t #f #t #f #t #t #t)) => #b111010100
(vector->bits '#(#f #t #t)) => 6
(vector->bits '#(#f #t #t #f)) => 6
(vector->bits '#(#f #f #t #t)) => 12
```

For positive integers, bits->list and list->bits are inverses in the sense of equal?, and so are bits->vector and vector->bits.

(bits bool ...)

Returns the integer coded by the bool arguments. The first argument is bit #0, the second argument is bit #1, and so on. Note that the result is never a negative integer.

```
(bits #t #f #t #f #t #t #t) => #b1110101
(bits #f #f #t #f #t #f #t #t #t) => #b111010100
```

(bitwise-fold proc seed i)

For each bit b of i from bit #0 (inclusive) to bit (integer-length i) (exclusive), proc is called as (proc b r), where r is the current accumulated result. The initial value of r is seed, and the value returned by proc becomes the next accumulated result. When the last bit has been processed, the final accumulated result becomes the result of bitwise-fold.

```
(bitwise-fold cons '() #b1010111) => (#t #f #t #f #t #t #t)
```

`(bitwise-for-each proc i)`

Repeatedly applies proc to the bits of i starting with bit #0 (inclusive) and ending with bit (integer-length i) (exclusive). The values returned by proc are discarded. Returns an unspecified value.

```
      (let ((count 0))
        (bitwise-for-each (lambda (b) (if b (set! count (+ count 1))))
                          #b1010111)
       count)
```

`(bitwise-unfold stop? mapper successor seed)`

Generates a non-negative integer bit by bit, starting with bit 0. If the result of applying stop? to the current state (whose initial value is seed) is true, return the currently accumulated bits as an integer. Otherwise, apply mapper to the current state to obtain the next bit of the result by interpreting a true value as a 1 bit and a false value as a 0 bit. Then get a new state by applying successor to the current state, and repeat this algorithm.

```
  (bitwise-unfold (lambda (i) (= i 10))
                  even?
                  (lambda (i) (+ i 1))
                  0)) => #b101010101
```

`(make-bitwise-generator i)`

Returns a SRFI 121 generator that generates all the bits of i starting with bit #0. Note that the generator is infinite.

```
(let ((g (make-bitwise-generator #b110)))
  (test #f (g))
  (test #t (g))
  (test #t (g))
  (test #f (g)))
```

(srfi srfi-158)

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 genn 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.

(bytevector-accumulator)

Returns an accumulator that, when invoked on a byte, adds that integer to a bytevector 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 bytevector.

(bytevector-accumulator! bytevector at)

Returns an accumulator that, when invoked on a byte, adds that byte to consecutive positions of bytevector 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 bytevector.

`(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.

`(srfi srfi-167)`

This is based on SRFI-167.

Abstract

This library describes an interface for an ordered key-value store. There is a memory backend and a wiredtiger backend.

This library contains four modules:

- `(srfi srfi-167 pack)`
- `(srfi srfi-167 engine)`
- `(srfi srfi-167 memory)`
- `(srfi srfi-167 wiredtiger)`

`(srfi srfi-167 pack)`

Abstract

This library allows to pack and unpack Scheme objects from and to bytevectors while preserving their natural order.

Reference

`(pack . args)` → `bytevector`

`(unpack bytevector)`

(srfi srfi-167 engine)

This is based on SRFI-167.

Abstract

This library describe a typeclass object for ordered key-value stores. It allows database abstraction to switch between ordered key-value store implementation easily.

Reference

`(make-engine ref set delete range-remove range prefix-range hook-on-transaction-begin hook-on-transaction-commit pack unpack)`

Return an engine record instance.

`(engine? obj)`

Return `#t` if `OBJ` is an engine record instance. Otherwise, it returns `#f`.

`(engine-ref engine)`

Return the procedure `okvs-ref`.

`(engine-set engine)`

Return the procedure `okvs-set!`.

`(engine-delete engine)`

Return the procedure `okvs-delete!`.

`(engine-range-remove engine)`

Return the procedure `okvs-range-remove!`.

`(engine-range engine)`

Return the procedure `okvs-range`.

`(engine-prefix-range engine)`

Return the procedure `okvs-prefix-range`.

`(engine-hook-on-transaction-begin engine)`

Return the procedure `okvs-hook-on-transaction-begin`.

`(engine-hook-on-transaction-commit engine)`

Return the procedure okvs-hook-on-transaction-commit.

`(engine-pack engine)`

Return a packing procedure that allows to encode some scheme types into bytevectors preserving their natural order. The supported Scheme types is implementation dependent.

`(engine-unpack engine)`

Return an unpacking procedure that will decode a bytevector encoded with the above `pack` procedure into a Scheme object.

`(srfi srfi-167 memory)`

This is based on SRFI-167.

Abstract

TODO

Reference

`(okvs-open home [config])`

TODO

`(okvs? obj)`

TODO

`(okvs-close okvs [config])`

TODO

`(make-default-state)`

TODO

`(okvs-transaction? obj)`

TODO

`(okvs-transaction-state transaction)`

TODO

`(okvs-in-transaction okvs proc [failure [success [make-sate [config]]]])`

TODO

`(okvs-ref okvs-or-transaction key)`

TODO

`(okvs-set! okvs-or-transaction key value)`

TODO

`(okvs-delete! okvs-or-transaction key)`

TODO

`(okvs-range-remove! okvs-or-transaction start-key start-include? end-key end-include?)`

TODO

‘(okvs-range okvs-or-transaction start-key start-include? end-key end-include? [CONFIG])

TODO

`(okvs-prefix-range okvs-or-transaction prefix [config])`

TODO

`(okvs-hook-on-transaction-begin okvs)`

TODO

`(okvs-hook-on-transaction-commit okvs)`

TODO

`(make-default-engine)`

Return an engine for the current okvs implementation.

`(srfi srfi-173)`

This is based on SRFI-173.

Abstract

This library describes a mechanism known as hooks. Hooks are a certain kind of extension point in a program that allows interleaving the execution of arbitrary code with the execution of the program without introducing any coupling between the two.

Reference

`(make-hook arity)`

Create a hook object for storing procedures of ARITY. The return value is a hook object.

`(hook? obj)`

Return #t if obj is a hook. Otherwise, it returns #f.

`(list->hook arity lst)`

Create a hook with the given procedures LST that must have an arity equal to ARITY. The return value is a hook object.

`(list->hook! hook lst)`

Replace procedures in HOOK by the procedures in LST. The return value is unspecified

`(hook-add! hook proc)`

Add the procedure PROC to the HOOK object. The return value is not specified. An implementation may check that the arity of PROC is equal to the arity of the HOOK.

`(hook-delete! hook proc)`

Delete the procedure PROC from the HOOK object. The return value is not specified.

`(hook-reset! hook)`

Remove all procedures from the HOOK object. The return value is not specified.

`(hook->list hook)`

Convert the list of procedures of HOOK object to a list.

`(hook-run hook . args)`

Apply all procedures from HOOK to the arguments ARGS. The order of the procedure application is not specified. The return value is not specified. The length of ARGS must be equal to the arity of the HOOK object.

`(arew stream)`

Abstract

Wanna be fast and functional streams.

Reference

`(list->stream lst)`

TODO

`(stream->list stream)`

TODO

`(stream-null)`

Return an empty stream.

`(stream-empty? stream)`

Return `#t` if the stream is empty. Otherwise, return `#f`.

`(stream-car stream)`

Return the first element of `STREAM`.

`(stream-map proc stream)`

Return a stream where `PROC` was applied to every value of `STREAM`.

`(stream-for-each proc stream)`

Apply `PROC` to every value of stream. The return value is undefined.

`(stream-filter predicate? stream)`

Return a stream of values from `STREAM` for which `PREDICATE?` return `#t`.

`(stream-apppend . streams)`

Return a stream made of the given `STREAMS`.

`(stream-concatenate stream)`

`STREAM` must be stream of stream. Return a stream of the values.

`(arew data json)`

Abstract

Provide a `(json->scm string)` procedure that parse a json string into a Scheme object.

Reference

`(json->scm string)`

Return a Scheme object representation of the JSON `STRING`. In case of error, return `#f`.

Arrays are represented as a list. JSON Objects are represented as an association list prefixed with a star symbol where keys are symbols.

`(arew data parser combinator)`

Abstract

parser combinators inspired from the following projects:

- https://epsil.github.io/gll/
- https://docs.racket-lang.org/parsack/index.html
- https://docs.racket-lang.org/megaparsack/
- https://git.dthompson.us/guile-parser-combinators.git
- https://gitlab.com/tampe/stis-parser

Reference

`(make-pseudo-xchar char)`

Make an extended character without line, column or offset information.

`(parse parser stream)`

Parse `STREAM` using `PARSER`. Return `#f` in case of error.

`(parse-any stream)`

Parser that succeed with anything in `STREAM`.

`(parse-char char)`

Parser that succeed with the given `CHAR`

`(parse-char-set char-set)`

Parser that succeed with the given `CHAR-SET` from `(scheme charset)`.

`(parse-each <parser> ...)` syntax

Parser that succeed if every parser succeed in sequence.

`(parse-either <parser> ...)` syntax

Parser that succeed when the first given parser succeed.

`(parse-lift proc parser)`

Apply `PROC` to the result of `PARSER`.

`(parse-maybe parser)`

If `PARSER` succeed return its result, otherwise return `#f` as result.

`(parse-one-or-more parser)`

Parser that succeed if `PARSER` succeed one or more time.

`(parse-only predicate? parser)`

Parser that succeed only when the result of `PARSER` returns `#t` when passed to `PREDICATE?`.

`(parse-return value)`

Parser that always succeed with `VALUE` as result.

`(parse-xstring string)`

Parser that succeed when `STRING` can be parsed as a sequence of extended characters.

`(parse-when predicate? parser)`

Parser that succeed when the next value returns `#t` when passed as argument to `PREDICATE?`. Return the result of `PARSER`.

`(parse-when* parser other)`

More general form of `parse-when`. If `PARSER` succeed, return the result of `OTHER` parser.

`(parse-unless predicate? parser)`

Parser that succeed unless the next value returns `#t` when passed as argument to `PREDICATE?`. Return the result of `PARSER`.

`(parse-unless* parser other)`

More general form of `parse-unless`. If `PARSER` fails, return the result of `OTHER` parser.

`(parse-xchar char)`

Parser that succeed if the next value is an extended character that is a `CHAR`.

`(parse-zero-or-more parser)`

Succeed if `PARSER` succeed zero or more time.

`(string->stream string)`

Create a stream of extended characters based on `STRING`.

`(xchar-char xchar)`

Return the character of `XCHAR`.

`(xchar? obj)`

Return `#t` if `OBJ` is an extended character. Otherwise, return `#f`.

`(arew data base lsm)`

Abstract

LSM is an embedded database library for key-value data, roughly similar in scope to Berkeley DB, LevelDB or KyotoCabinet. Both keys and values are specified and stored as byte arrays. Duplicate keys are not supported. Keys are always sorted in `memcmp()` order. LSM supports the following operations for the manipulation and query of database data:

- Writing a new key and value into the database.
- Deleting an existing key from the database.
- Deleting a range of keys from the database.
- Querying the database for a specific key.
- Iterating through a range of database keys (either forwards or backwards).

Other salient features are:

- A single-writer/multiple-reader MVCC based transactional concurrency model. SQL style nested sub-transactions are supported. Clients may concurrently access a single LSM database from within a single process or multiple application processes.
- An entire database is stored in a single file on disk.
- Data durability in the face of application or power failure. LSM may optionally use a write-ahead log file when writing to the database to ensure committed transactions are not lost if an application or power failure occurs.

See sqlite's lsm extension documentation.

Reference

`(lsm-new)`

Open a database connection handle.

`(lsm-close db)`

Close a database connection handle.

`(lsm-config db config value)`

Configuring a database connection.

TODO: document configuration.

`(lsm-open db filename)`

Connect to a database.

`(lsm-begin db level)`

Open a transaction or sub-transaction. To open a top-level transaction pass `1` as `LEVEL`. Passing `0` as `LEVEL` is no-op.

`(lsm-commit db level)`

Commit transaction and any sub-transactions. A successfull call to `lsm-commit` ensures that there are at most `LEVEL` nested transactions open. To commit a top-level transaction, pass `0` as `LEVEL`. To commit all sub-transactions inside the main transaction, pass `1` as `LEVEL`.

`(lsm-rollback db level)`

Roll back transaction and sub-transactions. A successful call to `lsm-rollback` restores the database to the state it was in when the `LEVEL`'th nested sub-transaction (if any) was first opened. And then closes transactions to ensure that there are at most `LEVEL` nested transactions open. Passing `0` as `LEVEL` rolls back and closes the top-level transaction. `LEVEL` equal to `1` also rolls back the top-level transaction, but leaves it open. `LEVEL` equal to `2` rolls back the sub-transaction nested directly inside the top-level transaction (and leaves it open).

`(lsm-insert db key value)`

Write a new value into the database where `KEY` and `VALUE` are bytevectors. If a value with a duplicate key already exists it is replaced.

`(lsm-delete db key)`

Delete a value from the database. No error is returned if the specified `KEY` does not exist in the database.

`(lsm-delete-range db key1 key2)`

TODO: Implement it.

`(lsm-cursor-open db)`

Open a cursor.

`(lsm-cursor-close cursor)`

Close a cursor.

`(lsm-cursor-seek cursor key seek)`

Position the cursor at `KEY` according to `SEEK` symbol.

`(lsm-cursor-first cursor)`

Position the cursor at the first value.

`(lsm-cursor-last cursor)`

Position the cursor at the last value.

`(lsm-cursor-next cursor)`

Move the cursor at the next value.

`(lsm-cursor-prev cursor)`

Move the cursor at the previous value.

`(lsm-cursor-valid? cursor)`

Determine whether or not the cursor currently points to a valid entry.

`(lsm-cursor-key cursor)`

Retrieve the key as a bytevector.

`(lsm-cursor-value cursor)`

Retrieve the value as a bytevector. ## `(cffi wiredtiger)`

`(connection? obj)`

Return `#t` if `OBJ` is a connection, otherwise `#f`.

`(connection-open path config)`

Open a new connection at `PATH` using `CONFIG`.

`(connection-close connection config)`

Close `CONNECTION` using `CONFIG`.

`(session? obj)`

Return `#t` if `OBJ` is a session, otherwise `#f`.

`(session-open connection config)`

Open a session against `CONNECTION` using `CONFIG`.

`(session-close session config)`

Close `SESSION` using `CONFIG`.

`(session-create session name)`

Create a table named `NAME` using `SESSION` with a bytevector column as key and value.

`(session-reset session)`

Reset `SESSION`.

`(session-transaction-begin session config)`

Begin transaction against `SESSION` using `CONFIG`.

`(session-transaction-commit session config)`

Commit transaction against `SESSION` using `CONFIG`.

`(session-transaction-rollback session config)`

Rollback transaction against `SESSION` using `CONFIG`.

`(cursor-open session uri config)`

Open a cursor against `SESSION` at `URI` using `CONFIG`.

`(cursor? cursor)`

Return `#t` if `CURSOR` is a cursor. Otherwise return `#f`.

`(cursor-close cursor)`

Close `CURSOR`.

`(cursor-uri cursor)`

Return `CURSOR`'s uri.

`(cursor-key-format cursor)`

Return `CURSOR`'s key format.

`(cursor-value-format cursor)`

Return `CURSOR`'s value format.

`(cursor-key-ref cursor)`

Return the key pointed by `CURSOR` as a bytevector.

`(cursor-value-ref cursor)`

Return the value pointed by `CURSOR` as a bytevector.

`(cursor-next? cursor)`

Move `CURSOR` to the next record, return `#t` if any. Otherwise it return `#f`.

`(cursor-prev? cursor)`

Move `CURSOR` to the previous record, return `#t` if any. Otherwise it return `#f`.

`(cursor-reset cursor)`

Reset `CURSOR`.

`(cursor-search? cursor key)`

Try to position `CURSOR` at `KEY`. Return `#t` if it succeed. Otherwise it return `#f`.

`(cursor-search-near cursor key)`

Try to position `CURSOR` near `KEY`. Return `'before`, `'exact` or `'after` depending on where the cursor is positioned. Otherwise it return `#f`.

`(cursor-insert cursor key value)`

Insert `KEY` and `VALUE` using `CURSOR`.

`(cursor-update cursor key value)`

Update `KEY` with `VALUE` using `CURSORS`.

`(cursor-remove cursor key)`

Remove record that has `KEY` as key using `CURSOR`.

`(arew network socket)`

Abstract

BSD Sockets

Reference

`(close fd)`

Close the socket described by `FD`.

`(socket domain type protocol)`

`DOMAIN` can be one the following symbol:

- `unspecified`
- `unix` or `local`
- `inet`
- `inet6`

`TYPE` can be one the following symbol:

- `stream`
- `datagram`
- `raw`
- `rdm`
- `seqpacket`
- `dccp`
- `packet`
- `close-on-exec`
- `non-blocking`

`PROTOCOL` can be one the following symbol:

- `ipv4`
- `icmp`
- `igmp`

- `ipip`
- `gcp`
- `egp`
- `pup`
- `udp`
- `idp`
- `tp`
- `dccp`
- `ipv6`
- `rsvp`
- `gre`
- `esp`
- `ah`
- `beetph`
- `encap`
- `pim`
- `comp`
- `sctp`
- `udplite`
- `mpls`
- `raw`

`(connect sockfd address)`

`SOCKFD` must be socket opened with `socket`. `ADDRESS` must be an association list with the following keys:

- `family` only `inet` value is supported
- `port` a number below 65536
- `address` a string describing an ipv4 address.

`(getaddrinfo node service hints)`

`NODE`, `SERVICE` and `HINTS` may be false. If `HINTS` is provided it must be an association list with the following keys:

- `flags`
- `family`
- `type`
- `protocol`

`flags` value can be one or more the following:

- `passive`
- `canonname`
- `v4mapped`
- `numerichost`

- `all`
- `addrconfig`
- `idn`
- `canonidn`
- `idn-allow-unassigned`
- `idn-use-std3-ascii-rules`

`family`, `type` and `protocol` are described above.

`(recv fd bytevector flags)`

`flags` can be `#f` one or more of the following symbols:

- `oob`
- `peek`
- `tryhard`
- `ctrunc`
- `proxy`
- `trunc`
- `dontwait`
- `eor`
- `waitall`
- `fin`
- `syn`
- `confirm`
- `rst`
- `errqueue`
- `nosignal`
- `more`
- `waitforone`
- `batch`
- `fastopen`
- `cmsg-cloexec`

It will return `-1` on error and `errno` is set. Otherwise it return the count bytes read.

`(recvfrom fd bytevector flags)`

`FLAGS` is described above. Return two values. The first value `#f` in case of error and `errno` can be used to retrieve the error. Otherwise it return the count bytes read along the address as an association list.

`(send fd bytevector flags)`

`FLAGS` is described above. In case of success return the count of bytes sent. Otherwise, it return `-1` and `errno` can be used to retrieve the error.

`(sendto fd bytevector flags address)`

`FLAGS` is described above. `address` must be an association list description of the destination.

`(accept sock)`

Return the file descriptor of the accepted connection. Otherwise raise an error.

`(bind sock address)`

`(fcntl sock command)`

`(fcntl! sock command flags)`

`(setsockopt socket level optname optval optlen)`

`(listen sock backlog)`

`(fd->port fd)`

Return a port based on `FD`.