Let's agree that *serif* fonts do not always carry boring stuff. And have a taste of it:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. — Revised 7 Report on the Algorithmic Language Scheme, Introduction.

Otherwise said, Scheme offers a minimalist core of powerful primitives upon which one can build abstractions to solve (real world) problems.

The Scheme universe is vast and prolific. As programming languages, Scheme dialects target various niches and implement various paradigms. Some of them are part of the de facto standards (namely RnRS and SRFIs).

The best-known paradigm of agreed-upon practices revolve around Functional Programming.

Scheme might be a dynamically typed language, but it can compete with its scions and siblings when performance matters.

Few programming languages can compete with Scheme when it comes to computer science whether it is Programming Language Theory, or Artificial Intelligence.

That being said, Scheme implementations might be missing some love. That's a good opportunity for you to learn something useful and give something back.

Or, like others, to make it your secret sauce.

Discourse

Tutorial

Standard Library

- `(scheme base)` R7RS-small
- `(scheme bitwise)`
- `(scheme box)`
- `(scheme bytevector)`
- `(scheme case-lambda)` R7RS-small
- `(scheme char)` R7RS-small
- `(scheme charset)`
- `(scheme comparator)`
- `(scheme complex)` R7RS-small
- `(scheme cxr)` R7RS-small
- `(scheme division)`
- `(scheme ephemeron)`
- `(scheme eval)` R7RS-small
- `(scheme file)` R7RS-small

- `(scheme fixnum)`
- `(scheme flonum)`
- `(scheme generator)`
- `(scheme hash-table)`
- `(scheme idque)`
- `(scheme ilist)`
- `(scheme inexact)` R7RS-small
- `(scheme lazy)` R7RS-small
- `(scheme list)`
- `(scheme list-queue)`
- `(scheme load)` R7RS-small
- `(scheme lseq)`
- `(scheme mapping)`
- `(scheme mapping-hash)`
- `(scheme process-context)` R7RS-small
- `(scheme r5rs)` R7RS-small
- `(scheme read)` R7RS-small
- `(scheme regex)`
- `(scheme repl)` R7RS-small
- `(scheme rlist)`
- `(scheme set)`
- `(scheme show)`
- `(scheme sort)`
- `(scheme stream)`
- `(scheme text)`
- `(scheme time)` R7RS-small
- `(scheme vector)`
- `(scheme write)` R7RS-small

Source, and single page files

You can find the source over the rainbow. There is available a single markdown file, and a single html file and a pdf;

LICENSE

 # A cheatsheet on Discourse.

The three gates of speech

Before you speak, let your words pass through three gates.

- At the first gate, ask yourself, is it true.
- At the second gate ask, is it necessary.
- At the third gate ask, is it kind.

Rogerian rhetoric

> 1. You should attempt to re-express your target's position so clearly, vividly, and fairly that your target says, “Thanks, I wish I'd thought of putting it that way.”
> 2. You should list any points of agreement (especially if they are not matters of general or widespread agreement).
> 3. You should mention anything you have learned from your target.
> 4. Only then are you permitted to say so much as a word of rebuttal or criticism.
>
> — Dennett's version of Rapoport's Rules

Argument Ranking

- - High-level generators - Disagreements that remain when everyone understands exactly what's being argued, and agrees on what all the evidence says, but have vague and hard-to-define reasons for disagreeing.
- - Operationalizing - Where both parties understand they're in a cooperative effort to fix exactly what they're arguing about.
- - Survey of evidence - Not trying to devastate the other person with a mountain of facts and start looking at the studies and arguments on both sides and figuring out what kind of complex picture they paint.

- - Disputing definitions - Argument hinges on the meaning of words, or whether something counts as a member of a category or not.
- - Single Studies - Better than scattered facts, proving they at least looked into the issue formally.
- - Demands for rigor - Attempts to demand that an opposing argument be held to such strict standards that nothing could possibly clear the bar.
- - Single Facts- One fact, which admittedly does support their argument, but presented as if it solves the debate in and of itself.
- - Gotchas - Short claims that purport to be devastating proof that one side can't possibly be right.
- - Social shaming- A demand for listeners to place someone outside the boundary of whom deserve to be heard.

"How to apologize: Quickly, specifically, sincerely."

— Kevin Kelly

Arguments

- **Ad baculum** : Argument relying on an appeal to fear or a threat.
- **Ad ignorantiam** : Argument relying on people's ignorance.
- **Ad populum** : Argument relying on sentimental weakness.
- **Ad verecundiam** : Argument relying on the the words of an "expert", or authority.
- **Ex silentio** : Argument relying on ignorance.
- **Ex nihilo** : An argument that bears no relation to the previous topic of discussion.
- **Non sequitur** : An inference that does not follow from established premises or evidence.

Responses

- **Akrasia** : State of acting against one's better judgment.
- **Connotation** : Emotional association with a word.
- **Intransigence** : Refusal to change one's views or to agree about something.
- **Inferential distance** : Gap between the background knowledge and epistemology of a person trying to explain an idea, and the background knowledge and epistemology of the person trying to understand it.
- **Straw man** : Creating a false or made up scenario and then attacking it. Painting your opponent with false colors only deflects the purpose of the argument.
- **Steel man** : To steelman is to address the strongest possible variant or the most charitable interpretation of an idea, rather than the most available phrasings.

- **Red herring** : A diversion from the active topic.
- **Rationalization** : Starts from a conclusion, and then works backward to arrive at arguments apparently favouring that conclusion. Rationalization argues for a side already selected.
- **Dogpiling** : A disagreement wherein one person says something wrong or offensive, and a large number of people comment in response to tell them how wrong they are, and continue to disparage the original commenter beyond any reasonable time limit.
- **Grandstanding** : An action that is intended to make people notice and admire you, behaving in a way that makes people pay attention to you instead of thinking about more important matters.
- **Whataboutism** : An attempt to discredit an opponent's position by charging them with hypocrisy without directly refuting or disproving their argument.
- **Dissensus** : The deliberate avoidance of consensus.

Beliefs

- **Belief** : The mental state in which an individual holds a proposition to be true.
- **Priors** : The beliefs an agent holds regarding a fact, hypothesis or consequence, before being presented with evidence.
- **Alief** : An independent source of emotional reaction which can coexist with a contradictory belief. Example The fear felt when a monster jumps out of the darkness in a scary movie is based on the alief that the monster is about to attack you, even though you believe that it cannot.
- **Proper belief** : Requires observations, gets updated upon encountering new evidence, and provides practical benefit in anticipated experience.
- **Improper belief** : Is a belief that isn't concerned with describing the territory. Note that the fact that a belief just happens to be true doesn't mean you're right to have it. If you buy a lottery ticket, certain that it's a winning ticket (for no reason), and it happens to be, believing that was still a mistake.
- **Belief in belief** : Where it is difficult to believe a thing, it is often much easier to believe that you ought to believe it. Were you to really believe and not just believe in belief, the consequences of error would be much more severe. When someone makes up excuses in advance, it would seem to require that belief, and belief in belief, have become unsynchronized.
- **A Priori** : Knowledge which we can be sure of without any empirical evidence(evidence from our senses). So, knowledge that you could realize if you were just a mind floating in a void unconnected to a body.

> "A leader is best when people barely know they exists, when their work is done, their aim fulfilled, people will say: we did it ourselves."
>
> — (Lao Tse), (Dao De Jing)

The first principle of Wikipedia etiquette has been said to be **Assume Good Faith**, also they **Be Bold, but not Reckless**.

Wrong discourse

- Answer: Jumping into a conversation with endless unapplicable, unrealistic or unrelated answers to the question.
- Question: Spouting accusations while cowardly hiding behind the claim of just asking questions, and ignoring the answers. Asking loaded questions.

Good discourse

- Answer: A clear and honest response to the central point of a question without an aggressive attempt to convince.
- Question: question asked with the intention to be fair, open, and honest, regardless of the outcome of the interaction.

Social rules are expected to be broken from time to time, in that regard they are different from a code of conduct.

Response Ranking

- - Central point - Commit to refute explicitly the central point.
- - Refutation - Argue a conflicting passage, explain why it's mistaken.
- - Counterargument - Contradict with added reasoning or evidence.
- - Contradiction - State the opposing case, what.
- - Responding to Tone - Responding to the author's tone, how.
- - Ad Hominem - Attacking the author directly, who.

Interaction Ranking

Discussion

- - Release - Initiating a discussion on the lessons learnt from a project.
- - Update - Presenting the recent development of a personal experience, ongoing event or work in progress.
- - Soapbox - Spontaneous and or enthusiastic posts about a general topic of interest or finding.

Low-Effort

- - Rant- Venting frustration publicly without explicitly looking to have a conversation about the matter.

- - Shitpost - Aggressively or ironically looking for the biggest reaction with the least effort possible. Includes subtoots and vague-posting.

Emotional Reaction

- **Seduction** - You are led to feel that the fulfillment of your dreams depends on your doing what the other is encouraging you to do.
- **Alignment** - The interests of the system are presented as fulfilling your emotional needs. You are led to feel that your survival, your viability in society or your very identity depends on your doing what the other is requiring of you.
- **Reduction** - Complex subjects are reduced to a single, emotionally charged issue.
- **Polarization** - Issues are presented in such a way that you are either right or wrong. You are told that any dialogue between different perspectives is suspect, dangerous or simply not permissible.
- **Marginalization** - You are made to feel that your own interests (or interests that run counter to the interests of the other) are inconsequential.
- **Framing** - The terms of a debate are set so that issues that threaten the system cannot be articulated or discussed. You are led to ignore aspects of the issue that may be vitally important to your own interests but are contrary to the interests of the other that is seeking to make you act in their interests.

Quotes

> "Kings speak for the realm, governors for the state, popes for the church. Indeed, the titled, as titled, cannot speak **with** annyone."
>
> — James P. Carse, Finite and Infinite Games

> "Instead of trying to prove your opponent wrong, try to see in what sense he might be right." — Robert Nozick, Anarchy, State, and Utopia

> "I don't argue: I just say what I know or what I believe, as the case may be." — John W. Cohan

> "You should mention anything you have learned from your target."

LICENSE

with “wrong”) and “faith” (replaced with “discourse”), a few other changes, see history for complete log. # Tutorial

Basics

Continuation

After reading this section you will be able to write basic Scheme programs. In particular, you will study:

- How to comment code
- How to write literals for builtin types
- How to call a procedure
- How to define a variable
- How to compare objects
- How to define a procedure

How to comment code

You can comment code with the semi-colon, that is `;`. Idiomatic code use two semi-colons:

```
;; Everything after one semi-colon is a comment.
```

The following sections will use two semi-colons with followed by an arrow `=>` to describe the return value.

How to write literals for builtin types

number

- Integers can be written as usual `42`
- Inexact reals can be written as usual `3.1415`
- There is more number types. It is called the Numerical tower

boolean

- false: `#f`
- true: `#t`

characters Characters can be written with their natural representation prefixed with `#\\`, for instance the character `x` is represented in Scheme code as follow:

```
#\x
```

string A string is written with double quotes, that is `"`, for instance:

```
"hello world"
```

symbol A symbol is most of the time written with a simple quote prefix, that is `'`. For instance:

```
'unique
```

pair A pair of the symbol `'pi` and the value `3.1415` can be written as:

```
'(pi . 3.1415)
```

list A list can be written as literals separated by one space and enclosed by parenthesis. For instance, the following list has three items:

```
'(unique "hello world" (pi . 3.1415))`
```

The first item is the symbol `'unique`, the second item is a string, the third item is a pair.

The empty list is written `'()`.

vector A vector looks somewhat like a list but without the explicit simple quote. It use a hash prefix. For instance, the following vector has three items:

```
#(unique "hello world" 42)
```

The first item is the symbol `'unique`, the second item is a string, the third item is a number.

bytevector A bytevector is like vector but can contain only bytes. It looks like a list of integers, prefixed with `#vu8`. For instance, the following bytevector has three bytes:

```
#vu8(0 42 255)
```

How to call a procedure

A procedure call looks like a list without the simple quote prefix.

The following describe the addition 21 and 21:

```
(+ 21 21) ;; => 42
```

It returns `42`. So does the following multiplication:

```
(* 21 2) ;; => 42
```

The first item is a procedure object. Most of the time, procedure names are made of letters separated with dashes. That usually called `kebab-case`.

Here is another procedure call:

```
(string-append "hello" " " "world") ;; => "hello world"
```

It will return a string `"hello world"`.

How to define a variable

The first kind of variables that you encountered are procedures, things like `+`, `*` or `string-append`.

Variables can also contain constants. You can use `define`:

```
(define %thruth 42)
```

The above code will create a variable called `%thruth` that contains `42`.

Look at this very complicated computation:

```
(+ %thruth 1 (* 2 647)) ;; => 1337
```

How to compare objects

Identity equivalence To compare by identity, in pratice, whether two object represent the same memory location, you can use the procedure `eq?`.

In the case where you are comparing symbols you can use the procedure `eq?`:

```
(eq? 'unique 'unique) ;; => #t
(eq? 'unique 'singleton) ;; => #f
```

Equivalence

If you do not know the type of the compared objects, or the objects can be of different types, you can use the procedure `equal?`:

```
(equal? #t "true") ;; => #f
```

The string `"true"` is not equivalent to the boolean `#t`.

It is rare to use `equal?`, because, usually, you know the type of the compared objects and the compared object have the same type.

Equivalence predicates

The astute reader might have recognized a pattern in the naming of the equivalence procedures `eq?` and `equal?`: both end with a question mark. That is a convention that all procedures that can only return a boolean should end with a question mark. Those are called *predicates*.

They are predicates for every builtin types. For instance string type has a string equivalence predicate written `string=?`:

```
(string=? "hello" "hello world" "hello, world!") ;; => #f
```

The predicate procedure `string=?` will return `#t` if all arguments are the same string, in the sense they contain the same characters.

How to define a procedure

The simplest procedure ever, is the procedure that takes no argument and returns itself:

```
(define (ruse)
  ruse)
```

The above is sugar syntax for the following:

```
(define ruse (lambda () ruse))
```

A procedure that takes no arguments is called a *thunk*. Indentation and the newline are cosmetic conventions. If you call the procedure `ruse`, it will return `ruse`:

```
(eq? ruse (ruse))
```

One can define a procedure that adds one as follow:

```
(define (add1 number)
  (+ number 1))
```

The predicate to compare numbers is `=`. Hence, the following:

```
(= 2006 (add1 2005)) ;; => #t
```

Mind the fact that it returns a new number. It does not mutate the value even if it is passed as a variable.

Let's imagine a procedure that appends a name to the string `"Hello"`. For instance, given `"Aziz"` or a variable containing `"Aziz"`, it will return `"Hello Aziz"`.

```
(define name "Aziz")

(define (say-hello name)
  (string-append "Hello " name))

(string=? "Hello Aziz" (say-hello name)) ;; => #t

;; XXX: the variable name still contains "Aziz"

(string=? name "Aziz")) ;; => #t
```

It does not matter for the callee whether the arguments are passed as variables or literals:

```
(string=? "Hello John"  (say-hello "John")) ;; => #t
```

Backtrack

In this section you learned:

- How to comment code using a semi-colon character `;`
- How to write literals for builtin types
  - integer: `42`
  - float: `3.1415`
  - symbol: `'unique`
  - string: `"hello world"`
  - pair: `(pi . 3.1415)`
  - list: `'(42 "hello world" (pi . 3.1415))`
  - vector: `#(42 "hello world" (pi . 3.1415))`
  - bytevector: `#vu8(1 42 255)`
- How to call a procedure `(string-append "hello " "Aziz")`
- How to define a variable `(define %thruth 42)`
- How to compare objects using their type specific predicates. For instance: `(string=? "hello" "hello")`
- How to define a procedure again using `define` with slightly different syntax `(define (add1 number) (+ number 1))`

Forward

Continuation

After reading this section you will be able to write more complex Scheme code. In particular you will study:

- How to create lexical bindings
- How to set a variable
- How to do a branch `if`
- How to create a new type
- How to write a named-let

How to create lexical bindings

Lexical bindings can be created with `let`, `let*`, `letrec` and `letrec*`. They have slightly different behaviors, but the same syntax:

```
(let (<binding> ...) <expression> ...)
```

Where `<binding>` looks like an association of a variable name with the initial value it is holding. For instance:

```
(let ((a 1)
      (b 2))
  (+ a b 3)) ;; => 6
```

The above `let` form will bind `a` to `1`, `b` to `2` and return the output of `(+ a b 3)` that is `6`.

How to set a variable

To change what a variable holds without overriding it or mutating the object contained in the varialbe, you can use `set!`. Mind the exclamation mark, it is a convention that forms that have a side-effect ends with a exclamation mark. For instance:

```
(define %thruth 42)

(display %truth)
(newline)

(set! %thruth 101)

(display %truth)
(newline)
```

How to do a branch `if`

Scheme `if` will consider false, only the object `#f`. Hence, one can do the following:

```
(if #t
  (display "true")
  (display "never executed"))
```

Similarly:

```
(if #f
  (display "never executed")
  (display "false"))
```

In particular, the number zero is true according to scheme `if`:

```
(if 0
  (display "zero is true")
  (display "never executed"))
```

If you want to check whether a value is zero you can use the predicate `zero?` like so:

```
(if (zero? %thruth)
   (display "%thruth is zero")
   (display "%thruth is not zero"))
```

Or the less idiomatic predicate `=`:

```
(if (= %truth 0)
  (display "%thruth is zero")
  (display "%thruth is not zero"))
```

How to create a new type

To create a new type you can use the macro `define-record-type`. For instance, in a todo list application, we will need an `<item>` type that can be defined as:

```
(define-record-type <item>
  (make-item title body status)
  item?
  (title item-title item-title!)
  (body item-body item-body!)
  (status item-status item-status!))
```

Where:

- `<item>` is the record name,
- `make-item` is the constructor of record instances,
- `item?` is the predicate that allows to tell whether an object is a `<item>` type,
- `title`, `body` and `status` are fields with their associated getters and setters. Setters ends with an exclamation mark. They will mutate the object. Setters are optional.

Here is an example use of the above `<item>` definition:

```
(define item (make-item "Learn Scheme" "The Scheme programming language is awesome, I should

;; To change the status, one can do the following:

(item-status! item 'wip)

;; to get the title, one can do the following:

(display (item-title item))
(newline)
```

How to write a named-let

A named-let allows to do recursion without going through the ceremony of defining a separate procedure. In pratice, it used in similar contexts such as `for` or `while` loop in other languages. Given the procedure `(cons item items)` that will return a new list with `ITEMS` as tail and `ITEM` as first item, study the following code:

```
(let loop ((index 0)
           (out '()))
  (if (= index 10)
      (display out)
      (loop (+ index 1) (cons index out))))
```

It is equivalent to the following:

```
(define (loop index out)
  (if (= index 10)
      (display out)
      (loop (+ index 1) (cons index out))))

(loop 0 '())
```

A named-let, look like a `let` form that can be used to bind variables prefixed with a name. Here is some pseudo-code that describe the syntax of the named-let form:

```
(let <name> (<binding> ...) expression ...))
```

So `<binding>` and `<expression>` are very similar to a `let`. `<name>` will be bound to a procedure that takes as many argument as there is `<binding>` and its body will be `<expression> ...`. It will be called with the associated objects in `<binding> ...`. `expression` can call `<name>` most likely in tail call position but not necessarly. If the named-let is not tail-recursive, it is also known to be a *grow the stack recursive call*. Another way to see the named-let is pseudo-code:

```
(define <name> (lambda <formals> <expression> ...))

(<name> <arguments> ...)
```

Where:

- `<formals>` are the variable names from `<binding> ...`
- `<arguments>` are the initial object bound in `<binding> ...`

That is all.

Backtrack

- How to create lexical bindings with `let`, `let*`, `letrec` and `letrec*`,
- How to set a variable using `(set! %thruth 42)`,
- How to do a `if` with `(if %thruth (display "That is true") (display "That is false"))`,
- How to create a new type using `define-record-type` that can look like:

```
(define-record-type <record-name>
  (make-record-name field0 ...)
```

```
  record-name?
  (field0 record-name-field0 record-name-field0!))
```

- How to write a named-let, for instance an infinite loop will look like:

```
(let loop ((index 0))
  (display index)
  (loop (+ index 1)))
```

Beyond

Continuation

After reading this section you will be able to create libraries.

Backtrack

Elements of Style

(scheme fixnum)

This is based on SRFI-143.

This library describes arithmetic procedures applicable to a limited range of exact integers only. These procedures are semantically similar to the corresponding generic-arithmetic procedures, but allow more efficient implementations.

fx-width

Bound to the value w that specifies the implementation-defined range. (R6RS fixnum-width is a procedure that always returns this value.)

fx-greatest

Bound to the value 2w-1-1, the largest representable fixnum. (R6RS greatest-fixnum is a procedure that always returns this value.)

fx-least

Bound to the value -2w-1, the smallest representable fixnum. (R6RS least-fixnum is a procedure that always returns this value.)

(fixnum? obj)

Returns #t if obj is an exact integer within the fixnum range, and #f otherwise.

(fx=? i ...)

Semantically equivalent to =.

(fx<? i ...)

Semantically equivalent to <.

(fx>? i ...)

Semantically equivalent to >.

(fx<=? i ...)

Semantically equivalent to <=.

(fx>=? i ...)

Semantically equivalent to >=.

(fxzero? i)

Semantically equivalent to zero?.

(fxpositive? i)

Semantically equivalent to positive?.

(fxnegative? i)

Semantically equivalent to negative?.

(fxodd? i)

Semantically equivalent to odd?.

(fxeven? i)

Semantically equivalent to even?.

(fxmax i j ...)

Semantically equivalent to max.

(fxmin i j ...)

Semantically equivalent to min.

(fx+ i j)

Semantically equivalent to +, but accepts exactly two arguments.

(fx- i j)

Semantically equivalent to -, but accepts exactly two arguments.

(fxneg i)

Semantically equivalent to -, but accepts exactly one argument.

(fx* i j)

Semantically equivalent to *, but accepts exactly two arguments.

(fxquotient i j)

Semantically equivalent to quotient.

(fxremainder i j)

Semantically equivalent to remainder.

(fxabs i)

Semantically equivalent to abs. In accordance with the fixnum rule, has undefined results when applied to fx-least.

(fxsquare i)

Semantically equivalent to square.

(fxsqrt i)

Semantically equivalent to exact-integer-sqrt (not sqrt).

(fx+/carry i j k)

Returns the two fixnum results of the following computation:

```
(let*-values (((s) (+ i j k))
              ((q r) (balanced/ s (expt 2 fx-width))))
  (values r q))
```

(fx-/carry i j k)

Returns the two fixnum results of the following computation:

```
(let*-values (((d) (- i j k))
              ((q r) (balanced/ d (expt 2 fx-width))))
  (values r q))
```

`(fx*/carry i j k)`

Returns the two fixnum results of the following computation:

```
(let*-values (((s) (+ (* i j) k))
              ((q r) (balanced/ s (expt 2 fx-width))))
  (values r q))
```

The balanced/ procedure is available in SRFI 141, and also in the R6RS base library under the name of div0-and-mod0. Bitwise operations

The following procedures are the fixnum counterparts of certain bitwise operations from SRFI 151 and the R6RS (rnrs arithmetic fixnums) library. In case of disagreement, SRFI 151 is preferred. The prefixes bitwise- and integer- are dropped for brevity and compatibility.

`(fxnot i)`

Semantically equivalent to bitwise-not.

`(fxand i ...)`

Semantically equivalent to bitwise-and.

`(fxior i ...)`

Semantically equivalent to bitwise-ior.

`(fxxor i ...)`

Semantically equivalent to bitwise-xor.

`(fxarithmetic-shift i count)`

Semantically equivalent to arithmetic-shift, except that it is an error for the absolute value of count to exceed w-1.

`(fxarithmetic-shift-left i count)`

The same as fxarithmetic-shift except that a negative value of count is an error. This is provided for additional efficiency.

`(fxarithmetic-shift-right i count)`

The same as fxarithmetic-shift except that a non-negative value of count specifies the number of bits to shift right, and a negative value is an error. This is provided for additional efficiency.

`(fxbit-count i)`

Semantically equivalent to SRFI 151 bit-count.

`(fxlength i)`

Semantically equivalent to integer-length.

`(fxif mask i j)`

Semantically equivalent to bitwise-if. It can be implemented as (fxior (fxand mask i) (fxand (fxnot mask) j))).

`(fxbit-set? index i)`

Semantically equivalent to SRFI 151 bit-set?, except that it is an error for index to be larger than or equal to fx-width.

`(fxcopy-bit index i boolean)`

Semantically equivalent to SRFI 151 copy-bit, except that it is an error for index to be larger than or equal to fx-width.

`(fxfirst-set-bit i)`

Semantically equivalent to first-set-bit.

`(fxbit-field i start end)`

Semantically equivalent to bit-field.

`(fxbit-field-rotate i count start end)`

Semantically equivalent to SRFI 151 bit-field-rotate.

`(fxbit-field-reverse i start end)`

Semantically equivalent to bit-field-reverse. # `(scheme stream)`

This is based on SRFI-41.

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.

Streams without memoization were first described by Peter Landin in 1965. Memoization became accepted as an essential feature of streams about a decade

later. Today, streams are the signature data type of functional programming languages such as Haskell.

This Scheme Request for Implementation describes two libraries for operating on streams: a canonical set of stream primitives and a set of procedures and syntax derived from those primitives that permits convenient expression of stream operations. They rely on facilities provided by R6RS, including libraries, records, and error reporting. To load both stream libraries, say:

```
(import (scheme stream))
```

stream-null

Stream-null is a promise that, when forced, is a single object, distinguishable from all other objects, that represents the null stream. Stream-null is immutable and unique.

(stream-cons object stream)

Stream-cons is a macro that accepts an object and a stream and creates a newly-allocated stream containing a promise that, when forced, is a stream-pair with the object in its stream-car and the stream in its stream-cdr. Stream-cons must be syntactic, not procedural, because neither object nor stream is evaluated when stream-cons is called. Since stream is not evaluated, when the stream-pair is created, it is not an error to call stream-cons with a stream that is not of type stream; however, doing so will cause an error later when the stream-cdr of the stream-pair is accessed. Once created, a stream-pair is immutable; there is no stream-set-car! or stream-set-cdr! that modifies an existing stream-pair. There is no dotted-pair or improper stream as with lists.

(stream? object)

Stream? is a procedure that takes an object and returns #t if the object is a stream and #f otherwise. If object is a stream, stream? does not force its promise. If (stream? obj) is #t, then one of (stream-null? obj) and (stream-pair? obj) will be #t and the other will be #f; if (stream? obj) is #f, both (stream-null? obj) and (stream-pair? obj) will be #f.

(stream-null? object)

Stream-null? is a procedure that takes an object and returns #t if the object is the distinguished null stream and #f otherwise. If object is a stream, stream-null? must force its promise in order to distinguish stream-null from stream-pair.

(stream-pair? object)

Stream-pair? is a procedure that takes an object and returns #t if the object is a stream-pair constructed by stream-cons and #f otherwise. If object is a stream, stream-pair? must force its promise in order to distinguish stream-null from stream-pair.

(stream-car stream)

Stream-car is a procedure that takes a stream and returns the object stored in the stream-car of the stream. Stream-car signals an error if the object passed to it is not a stream-pair. Calling stream-car causes the object stored there to be evaluated if it has not yet been; the object's value is cached in case it is needed again.

'(stream-cdr stream)

Stream-cdr is a procedure that takes a stream and returns the stream stored in the stream-cdr of the stream. Stream-cdr signals an error if the object passed to it is not a stream-pair. Calling stream-cdr does not force the promise containing the stream stored in the stream-cdr of the stream.

(stream-lambda args body)

Stream-lambda creates a procedure that returns a promise to evaluate the body of the procedure. The last body expression to be evaluated must yield a stream. As with normal lambda, args may be a single variable name, in which case all the formal arguments are collected into a single list, or a list of variable names, which may be null if there are no arguments, proper if there are an exact number of arguments, or dotted if a fixed number of arguments is to be followed by zero or more arguments collected into a list. Body must contain at least one expression, and may contain internal definitions preceding any expressions to be evaluated.

```
(define strm123
  (stream-cons 1
    (stream-cons 2
      (stream-cons 3
        stream-null))))

(stream-car strm123)   1

(stream-car (stream-cdr strm123)   2

(stream-pair?
  (stream-cdr
    (stream-cons (/ 1 0) stream-null)))   #f
```

```
(stream? (list 1 2 3))   #f

(define iter
  (stream-lambda (f x)
    (stream-cons x (iter f (f x)))))

(define nats (iter (lambda (x) (+ x 1)) 0))

(stream-car (stream-cdr nats))   1

(define stream-add
  (stream-lambda (s1 s2)
    (stream-cons
      (+ (stream-car s1) (stream-car s2))
      (stream-add (stream-cdr s1)
                  (stream-cdr s2)))))

(define evens (stream-add nats nats))

(stream-car evens)   0

(stream-car (stream-cdr evens))   2

(stream-car (stream-cdr (stream-cdr evens)))   4
```

(define-stream (name args) body) syntax

Define-stream creates a procedure that returns a stream, and may appear anywhere a normal define may appear, including as an internal definition, and may have internal definitions of its own, including other define-streams. The defined procedure takes arguments in the same way as stream-lambda. Define-stream is syntactic sugar on stream-lambda; see also stream-let, which is also a sugaring of stream-lambda.

A simple version of stream-map that takes only a single input stream calls itself recursively:

```
(define-stream (stream-map proc strm)
  (if (stream-null? strm)
      stream-null
      (stream-cons
        (proc (stream-car strm))
        (stream-map proc (stream-cdr strm))))))
```

(list->stream list-of-objects)

[ ] → { }

List->stream takes a list of objects and returns a newly-allocated stream containing in its elements the objects in the list. Since the objects are given in a list, they are evaluated when list->stream is called, before the stream is created. If the list of objects is null, as in (list->stream '()), the null stream is returned. See also stream.

```
(define strm123 (list->stream '(1 2 3)))

; fails with divide-by-zero error
(define s (list->stream (list 1 (/ 1 0) -1)))
```

(port->stream [port])

port → {char}

Port->stream takes a port and returns a newly-allocated stream containing in its elements the characters on the port. If port is not given it defaults to the current input port. The returned stream has finite length and is terminated by stream-null.

It looks like one use of port->stream would be this:

```
(define s ;wrong!
  (with-input-from-file filename
    (lambda () (port->stream))))
```

But that fails, because with-input-from-file is eager, and closes the input port prematurely, before the first character is read. To read a file into a stream, say:

```
(define-stream (file->stream filename)
  (let ((p (open-input-file filename)))
    (stream-let loop ((c (read-char p)))
      (if (eof-object? c)
          (begin (close-input-port p)
                 stream-null)
          (stream-cons c
            (loop (read-char p)))))))
```

(stream object ...)

Stream is syntax that takes zero or more objects and creates a newly-allocated stream containing in its elements the objects, in order. Since stream is syntactic, the objects are evaluated when they are accessed, not when the stream is created. If no objects are given, as in (stream), the null stream is returned. See also list->stream.

```
(define strm123 (stream 1 2 3))
```

```
; (/ 1 0) not evaluated when stream is created
(define s (stream 1 (/ 1 0) -1))
```

(stream->list [n] stream)

nat × { } → [ ]

Stream->list takes a natural number n and a stream and returns a newly-allocated list containing in its elements the first n items in the stream. If the stream has less than n items all the items in the stream will be included in the returned list. If n is not given it defaults to infinity, which means that unless stream is finite stream->list will never return.

```
(stream->list 10
  (stream-map (lambda (x) (* x x))
    (stream-from 0)))
   (0 1 4 9 16 25 36 49 64 81)
```

(stream-append stream ...)

{ } … → { }

Stream-append returns a newly-allocated stream containing in its elements those elements contained in its input streams, in order of input. If any of the input streams is infinite, no elements of any of the succeeding input streams will appear in the output stream; thus, if x is infinite, (stream-append x y) x. See also stream-concat.

Quicksort can be used to sort a stream, using stream-append to build the output; the sort is lazy; so if only the beginning of the output stream is needed, the end of the stream is never sorted.

```
(define-stream (qsort lt? strm)
  (if (stream-null? strm)
      stream-null
      (let ((x (stream-car strm))
            (xs (stream-cdr strm)))
        (stream-append
          (qsort lt?
            (stream-filter
              (lambda (u) (lt? u x))
              xs))
          (stream x)
          (qsort lt?
            (stream-filter
              (lambda (u) (not (lt? u x)))
              xs))))))
```

Note also that, when used in tail position as in qsort, stream-append does not suffer the poor performance of append on lists. The list version of append requires re-traversal of all its list arguments except the last each time it is called. But stream-append is different. Each recursive call to stream-append is suspended; when it is later forced, the preceding elements of the result have already been traversed, so tail-recursive loops that produce streams are efficient even when each element is appended to the end of the result stream. This also implies that during traversal of the result only one promise needs to be kept in memory at a time.

(stream-concat stream)

{{ }} ... → { }

Stream-concat takes a stream consisting of one or more streams and returns a newly-allocated stream containing all the elements of the input streams. If any of the streams in the input stream is infinite, any remaining streams in the input stream will never appear in the output stream. See also stream-append.

```
(stream->list
  (stream-concat
    (stream
      (stream 1 2) (stream) (stream 3 2 1))))
   (1 2 3 2 1)
```

The permutations of a finite stream can be determined by interleaving each element of the stream in all possible positions within each permutation of the other elements of the stream. Interleave returns a stream of streams with x inserted in each possible position of yy:

```
(define-stream (interleave x yy)
  (stream-match yy
    (() (stream (stream x)))
    ((y . ys)
      (stream-append
        (stream (stream-cons x yy))
        (stream-map
          (lambda (z) (stream-cons y z))
          (interleave x ys))))))

(define-stream (perms xs)
  (if (stream-null? xs)
      (stream (stream))
      (stream-concat
        (stream-map
          (lambda (ys)
            (interleave (stream-car xs) ys))
          (perms (stream-cdr xs))))))
```

(stream-constant object ...)

... → { }

Stream-constant takes one or more objects and returns a newly-allocated stream containing in its elements the objects, repeating the objects in succession forever.

```
(stream-constant 1)   1 1 1 ...


(stream-constant #t #f)   #t #f #t #f #t #f ...
```

(stream-drop n stream) procedure

nat × { } → { }

Stream-drop returns the suffix of the input stream that starts at the next element after the first n elements. The output stream shares structure with the input stream; thus, promises forced in one instance of the stream are also forced in the other instance of the stream. If the input stream has less than n elements, stream-drop returns the null stream. See also stream-take.

```
(define (stream-split n strm)
  (values (stream-take n strm)
          (stream-drop n strm)))
```

(stream-drop-while pred? stream)

(→ boolean) × { } → { }

Stream-drop-while returns the suffix of the input stream that starts at the first element x for which (pred? x) is #f. The output stream shares structure with the input stream. See also stream-take-while.

Stream-unique creates a new stream that retains only the first of any subsequences of repeated elements.

```
(define-stream (stream-unique eql? strm)
  (if (stream-null? strm)
      stream-null
      (stream-cons (stream-car strm)
        (stream-unique eql?
          (stream-drop-while
            (lambda (x)
              (eql? (stream-car strm) x))
            strm)))))
```

(stream-filter pred? stream)

(→ boolean) × { } → { }

Stream-filter returns a newly-allocated stream that contains only those elements x of the input stream for which (pred? x) is non-#f.

```
(stream-filter odd? (stream-from 0))
    1 3 5 7 9 ...
```

(stream-fold proc base stream)

(× →) × × { } →

Stream-fold applies a binary procedure to base and the first element of stream to compute a new base, then applies the procedure to the new base and the next element of stream to compute a succeeding base, and so on, accumulating a value that is finally returned as the value of stream-fold when the end of the stream is reached. Stream must be finite, or stream-fold will enter an infinite loop. See also stream-scan, which is similar to stream-fold, but useful for infinite streams. For readers familiar with other functional languages, this is a left-fold; there is no corresponding right-fold, since right-fold relies on finite streams that are fully-evaluated, at which time they may as well be converted to a list.

Stream-fold is often used to summarize a stream in a single value, for instance, to compute the maximum element of a stream.

```
(define (stream-maximum lt? strm)
  (stream-fold
    (lambda (x y) (if (lt? x y) y x))
    (stream-car strm)
    (stream-cdr strm)))
```

Sometimes, it is useful to have stream-fold defined only on non-null streams:

```
(define (stream-fold-one proc strm)
  (stream-fold proc
    (stream-car strm)
    (stream-cdr strm)))
```

Stream-minimum can then be defined as:

```
(define (stream-minimum lt? strm)
  (stream-fold-one
    (lambda (x y) (if (lt? x y) x y))
    strm))
```

Stream-fold can also be used to build a stream:

```
(define-stream (isort lt? strm)
    (define-stream (insert strm x)
      (stream-match strm
        (() (stream x))
        ((y . ys)
          (if (lt? y x)
```

```
            (stream-cons y (insert ys x))
            (stream-cons x strm)))))
    (stream-fold insert stream-null strm))
```

(stream-for-each proc stream ...)

(× × ...) × { } × { } ...

Stream-for-each applies a procedure element-wise to corresponding elements of the input streams for its side-effects; it returns nothing. Stream-for-each stops as soon as any of its input streams is exhausted.

The following procedure displays the contents of a file:

```
(define (display-file filename)
  (stream-for-each display
    (file->stream filename)))
```

(stream-from first [step])

number × number → {number}

Stream-from creates a newly-allocated stream that contains first as its first element and increments each succeeding element by step. If step is not given it defaults to 1. First and step may be of any numeric type. Stream-from is frequently useful as a generator in stream-of expressions. See also stream-range for a similar procedure that creates finite streams.

Stream-from could be implemented as (stream-iterate (lambda (x) (+ x step)) first).

```
(define nats (stream-from 0))   0 1 2 ...

(define odds (stream-from 1 2))   1 3 5 ...
```

(stream-iterate proc base)

(→) × → { }

Stream-iterate creates a newly-allocated stream containing base in its first element and applies proc to each element in turn to determine the succeeding element. See also stream-unfold and stream-unfolds.

```
(stream-iterate (lambda (x) (+ x 1)) 0)
   0 1 2 3 4 ...

(stream-iterate (lambda (x) (* x 2)) 1)
   1 2 4 8 16 ...
```

Given a seed between 0 and 232, exclusive, the following expression creates a stream of pseudo-random integers between 0 and 232, exclusive, beginning with seed, using the method described by Stephen Park and Keith Miller:

```
(stream-iterate
  (lambda (x) (modulo (* x 16807) 2147483647))
  seed)
```

Successive values of the continued fraction shown below approach the value of the "golden ratio" 1.618:

Continued fraction

The fractions can be calculated by the stream

```
(stream-iterate (lambda (x) (+ 1 (/ x))) 1)
```

(stream-length stream)

{ } → nat

Stream-length takes an input stream and returns the number of elements in the stream; it does not evaluate its elements. Stream-length may only be used on finite streams; it enters an infinite loop with infinite streams.

```
(stream-length strm123)  3
```

(stream-let tag ((var expr) ...) body) syntax

Stream-let creates a local scope that binds each variable to the value of its corresponding expression. It additionally binds tag to a procedure which takes the bound variables as arguments and body as its defining expressions, binding the tag with stream-lambda. Tag is in scope within body, and may be called recursively. When the expanded expression defined by the stream-let is evaluated, stream-let evaluates the expressions in its body in an environment containing the newly-bound variables, returning the value of the last expression evaluated, which must yield a stream.

Stream-let provides syntactic sugar on stream-lambda, in the same manner as normal let provides syntactic sugar on normal lambda. However, unlike normal let, the tag is required, not optional, because unnamed stream-let is meaningless.

Stream-member returns the first stream-pair of the input strm with a stream-car x that satisfies (eql? obj x), or the null stream if x is not present in strm.

```
(define-stream (stream-member eql? obj strm)
  (stream-let loop ((strm strm))
    (cond ((stream-null? strm) strm)
          ((eql? obj (stream-car strm)) strm)
          (else (loop (stream-cdr strm))))))
```

(stream-map proc stream ...)

(× … →) × { } × { } … → { }

Stream-map applies a procedure element-wise to corresponding elements of the input streams, returning a newly-allocated stream containing elements that are the results of those procedure applications. The output stream has as many elements as the minimum-length input stream, and may be infinite.

```
(define (square x) (* x x))

(stream-map square (stream 9 3))   81 9

(define (sigma f m n)
  (stream-fold + 0
    (stream-map f (stream-range m (+ n 1)))))

(sigma square 1 100)   338350
```

In some functional languages, stream-map takes only a single input stream, and stream-zipwith provides a companion function that takes multiple input streams.

(stream-match stream clause ...) syntax

Stream-match provides the syntax of pattern-matching for streams. The input stream is an expression that evaluates to a stream. Clauses are of the form (pattern [fender] expr), consisting of a pattern that matches a stream of a particular shape, an optional fender that must succeed if the pattern is to match, and an expression that is evaluated if the pattern matches. There are four types of patterns:

- () — Matches the null stream.
- (pat0 pat1 …) — Matches a finite stream with length exactly equal to the number of pattern elements.
- (pat0 pat1 … . patrest) — Matches an infinite stream, or a finite stream with length at least as great as the number of pattern elements before the literal dot.
- pat — Matches an entire stream. Should always appear last in the list of clauses; it's not an error to appear elsewhere, but subsequent clauses could never match.

Each pattern element pati may be either:

- An identifier — Matches any stream element. Additionally, the value of the stream element is bound to the variable named by the identifier, which is in scope in the fender and expression of the corresponding clause. Each identifier in a single pattern must be unique.

- A literal underscore — Matches any stream element, but creates no bindings.

The patterns are tested in order, left-to-right, until a matching pattern is found; if fender is present, it must evaluate as non-#f for the match to be successful. Pattern variables are bound in the corresponding fender and expression. Once the matching pattern is found, the corresponding expression is evaluated and returned as the result of the match. An error is signaled if no pattern matches the input stream.

Stream-match is often used to distinguish null streams from non-null streams, binding head and tail:

```
(define (len strm)
  (stream-match strm
    (() 0)
    ((head . tail) (+ 1 (len tail)))))
```

Fenders can test the common case where two stream elements must be identical; the else pattern is an identifier bound to the entire stream, not a keyword as in cond.

```
(stream-match strm
  ((x y . _) (equal? x y) 'ok)
  (else 'error))
```

A more complex example uses two nested matchers to match two different stream arguments; (stream-merge lt? . strms) stably merges two or more streams ordered by the lt? predicate:

```
(define-stream (stream-merge lt? . strms)
  (define-stream (merge xx yy)
    (stream-match xx (() yy) ((x . xs)
      (stream-match yy (() xx) ((y . ys)
        (if (lt? y x)
            (stream-cons y (merge xx ys))
            (stream-cons x (merge xs yy))))))))
  (stream-let loop ((strms strms))
    (cond ((null? strms) stream-null)
          ((null? (cdr strms)) (car strms))
          (else (merge (car strms)
                       (apply stream-merge lt?
                         (cdr strms)))))))
```

(stream-of expr clause ...) syntax

Stream-of provides the syntax of stream comprehensions, which generate streams by means of looping expressions. The result is a stream of objects of the type returned by expr. There are four types of clauses:

- (var in stream-expr) — Loop over the elements of stream-expr, in order from the start of the stream, binding each element of the stream in turn to var. Stream-from and stream-range are frequently useful as generators for stream-expr.
- (var is expr) — Bind var to the value obtained by evaluating expr.
- (pred? expr) — Include in the output stream only those elements x for which (pred? x) is non-#f.

The scope of variables bound in the stream comprehension is the clauses to the right of the binding clause (but not the binding clause itself) plus the result expression.

When two or more generators are present, the loops are processed as if they are nested from left to right; that is, the rightmost generator varies fastest. A consequence of this is that only the first generator may be infinite and all subsequent generators must be finite. If no generators are present, the result of a stream comprehension is a stream containing the result expression; thus, (stream-of 1) produces a finite stream containing only the element 1.

```
(stream-of (* x x)
  (x in (stream-range 0 10))
  (even? x))
   0 4 16 36 64

(stream-of (list a b)
  (a in (stream-range 1 4))
  (b in (stream-range 1 3)))
   (1 1) (1 2) (2 1) (2 2) (3 1) (3 2)

(stream-of (list i j)
  (i in (stream-range 1 5))
  (j in (stream-range (+ i 1) 5)))
   (1 2) (1 3) (1 4) (2 3) (2 4) (3 4)
```

(stream-range first past [step])

number × number × number → {number}

Stream-range creates a newly-allocated stream that contains first as its first element and increments each succeeding element by step. The stream is finite and ends before past, which is not an element of the stream. If step is not given it defaults to 1 if first is less than past and -1 otherwise. First, past and step may be of any numeric type. Stream-range is frequently useful as a generator in stream-of expressions. See also stream-from for a similar procedure that creates infinite streams.

```
(stream-range 0 10)   0 1 2 3 4 5 6 7 8 9
```

```
(stream-range 0 10 2) → 0 2 4 6 8
```

Successive elements of the stream are calculated by adding step to first, so if any of first, past or step are inexact, the length of the output stream may differ from (ceiling (- (/ (- past first) step) 1).

(stream-ref stream n)

{ } × nat →

Stream-ref returns the nth element of stream, counting from zero. An error is signaled if n is greater than or equal to the length of stream.

```
(define (fact n)
  (stream-ref
    (stream-scan * 1 (stream-from 1))
    n))
```

(stream-reverse stream)

{ } → { }

Stream-reverse returns a newly-allocated stream containing the elements of the input stream but in reverse order. Stream-reverse may only be used with finite streams; it enters an infinite loop with infinite streams. Stream-reverse does not force evaluation of the elements of the stream.

```
> (define s (stream 1 (/ 1 0) -1))
> (define r (stream-reverse s))
> (stream-ref r 0)
> (stream-ref r 2)
1
> (stream-ref r 1)
error: division by zero
```

(stream-scan proc base stream)

(× →) × × { } → { }

Stream-scan accumulates the partial folds of an input stream into a newly-allocated output stream. The output stream is the base followed by (stream-fold proc base (stream-take i stream)) for each of the first i elements of stream.

```
(stream-scan + 0 (stream-from 1))
   (stream 0 1 3 6 10 15 ...)

(stream-scan * 1 (stream-from 1))
   (stream 1 1 2 6 24 120 ...)
```

(stream-take n stream)

nat × { } → { }

Stream-take takes a non-negative integer n and a stream and returns a newly-allocated stream containing the first n elements of the input stream. If the input stream has less than n elements, so does the output stream. See also stream-drop.

Mergesort splits a stream into two equal-length pieces, sorts them recursively and merges the results:

```
(define-stream (msort lt? strm)
  (let* ((n (quotient (stream-length strm) 2))
         (ts (stream-take n strm))
         (ds (stream-drop n strm)))
    (if (zero? n)
        strm
        (stream-merge lt?
          (msort < ts) (msort < ds)))))
```

(stream-take-while pred? stream)

(→ boolean) × { } → { }

Stream-take-while takes a predicate and a stream and returns a newly-allocated stream containing those elements x that form the maximal prefix of the input stream for which (pred? x) is non-#f. See also stream-drop-while.

```
(stream-car
  (stream-reverse
    (stream-take-while
      (lambda (x) (< x 1000))
        primes)))   997
```

(stream-unfold map pred? gen base)

(→) × (→ boolean) × (→) × → { }

Stream-unfold is the fundamental recursive stream constructor. It constructs a stream by repeatedly applying gen to successive values of base, in the manner of stream-iterate, then applying map to each of the values so generated, appending each of the mapped values to the output stream as long as (pred? base) is non-#f. See also stream-iterate and stream-unfolds.

The expression below creates the finite stream 0 1 4 9 16 25 36 49 64 81. Initially the base is 0, which is less than 10, so map squares the base and the mapped value becomes the first element of the output stream. Then gen increments the base by 1, so it becomes 1; this is less than 10, so map squares the new base and 1 becomes the second element of the output stream. And so on, until the base

becomes 10, when pred? stops the recursion and stream-null ends the output stream.

```
(stream-unfold
  (lambda (x) (expt x 2)) ; map
  (lambda (x) (< x 10))   ; pred?
  (lambda (x) (+ x 1))    ; gen
  0)                      ; base
```

(stream-unfolds proc seed)

(→ (values × …)) × → (values { } …)

Stream-unfolds returns n newly-allocated streams containing those elements produced by successive calls to the generator proc, which takes the current seed as its argument and returns n+1 values

```
(proc seed → seed result0 ... resultn-1
```

where the returned seed is the input seed to the next call to the generator and resulti indicates how to produce the next element of the ith result stream:

- (value) — value is the next car of the result stream
- #f — no value produced by this iteration of the generator proc for the result stream
- () — the end of the result stream

It may require multiple calls of proc to produce the next element of any particular result stream. See also stream-iterate and stream-unfold.

Stream-unfolds is especially useful when writing expressions that return multiple streams. For instance, (stream-partition pred? strm) is equivalent to

```
(values
  (stream-filter pred? strm)
  (stream-filter
    (lambda (x) (not (pred? x))) strm))
```

but only tests pred? once for each element of strm.

```
(define (stream-partition pred? strm)
  (stream-unfolds
    (lambda (s)
      (if (stream-null? s)
          (values s '() '())
          (let ((a (stream-car s))
                (d (stream-cdr s)))
            (if (pred? a)
                (values d (list a) #f)
                (values d #f (list a))))))
    strm))
```

```
(call-with-values
  (lambda ()
    (stream-partition odd?
      (stream-range 1 6)))
  (lambda (odds evens)
    (list (stream->list odds)
          (stream->list evens))))
   ((1 3 5) (2 4))
```

(stream-zip stream ...)

{ } × { } × … → {[…]}

Stream-zip takes one or more input streams and returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input streams. The output stream is as long as the shortest input stream, if any of the input streams is finite, or is infinite if all the input streams are infinite.

A common use of stream-zip is to add an index to a stream, as in (stream-finds eql? obj strm), which returns all the zero-based indices in strm at which obj appears; (stream-find eql? obj strm) returns the first such index, or #f if obj is not in strm.

```
(define-stream (stream-finds eql? obj strm)
  (stream-of (car x)
    (x in (stream-zip (stream-from 0) strm))
    (eql? obj (cadr x))))

(define (stream-find eql? obj strm)
  (stream-car
    (stream-append
      (stream-finds eql? obj strm)
      (stream #f))))

(stream-find char=? #\l
  (list->stream
    (string->list "hello")))   2

(stream-find char=? #\l
  (list->stream
    (string->list "goodbye")))   #f
```

Stream-find is not as inefficient as it looks; although it calls stream-finds, which finds all matching indices, the matches are computed lazily, and only the first match is needed for stream-find. # `(scheme comparator)`

This library is based on SRFI-128.

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.

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

`(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.

`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.

`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.

`(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.

`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.

`(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.

`(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.

(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.

(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.

(=? 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.

(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.

`(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.

`(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.
`(scheme set)`

This library is based on SRFI-113.

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.

`(set comparator element ... )`

Returns a newly allocated empty set. The comparator argument is a SRFI 114 comparator, which is used to control and distinguish the elements of the set. The elements are used to initialize the set.

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

Create a newly allocated set as if by set using comparator. If the result of applying the predicate stop? to seed is true, return the set. Otherwise, apply the procedure mapper to seed. The value that mapper returns is added to the set. Then get a new seed by applying the procedure successor to seed, and repeat this algorithm. Predicates

`(set? obj)`

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

`(set-contains? set element)`

Returns #t if element is a member of set and #f otherwise.

`(set-empty? set)`

Returns #t if set has no elements and #f otherwise.

`(set-disjoint? set1 set2)`

Returns #t if set1 and set2 have no elements in common and #f otherwise.

`(set-member set element default)`

Returns the element of set that is equal, in the sense of set's equality predicate, to element. If element is not a member of set, default is returned.

`(set-element-comparator set)`

Returns the comparator used to compare the elements of set.

`(set-adjoin set element ...)`

The set-adjoin procedure returns a newly allocated set that uses the same comparator as set and contains all the values of set, and in addition each element unless it is already equal (in the sense of the comparator) to one of the existing or newly added members. It is an error to add an element to set that does not return #t when passed to the type test procedure of the comparator.

`(set-adjoin! set element ...)`

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

`(set-replace set element)`

The set-replace procedure returns a newly allocated set that uses the same comparator as set and contains all the values of set except as follows: If element is equal (in the sense of set's comparator) to an existing member of set, then that member is omitted and replaced by element. If there is no such element in set, then set is returned unchanged.

`(set-replace! set element)`

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

`(set-delete set element ...)`

`(set-delete! set element ...)`

`(set-delete-all set element-list)`

`(set-delete-all! set element-list)`

The set-delete procedure returns a newly allocated set containing all the values of set except for any that are equal (in the sense of set's comparator) to one or more of the elements. Any element that is not equal to some member of the set is ignored.

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

The set-delete-all and set-delete-all! procedures are the same as set-delete and set-delete!, except that they accept a single argument which is a list of elements to be deleted.

`(set-search! set element failure success)`

The set is searched for element. 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 element is found, then the success procedure is tail-called with the matching element of set and two continuations, update and remove, and is expected to tail-call one of them.

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

- Invoking (insert obj) causes element to be inserted into set.
- Invoking (ignore obj) causes set to remain unchanged.
- Invoking (update new-element obj) causes new-element to be inserted into set in place of element.

- Invoking (remove obj) causes the matching element of set to be removed from it.

In all cases, two values are returned: the possibly updated set and obj.

(set-size set)

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

(set-find predicate set failure)

Returns an arbitrarily chosen element of set that satisfies predicate, or the result of invoking failure with no arguments if there is none.

(set-count predicate set)

Returns the number of elements of set that satisfy predicate as an exact integer.

(set-any? predicate set)

Returns #t if any element of set satisfies predicate, or #f otherwise. Note that this differs from the SRFI 1 analogue because it does not return an element of the set.

(set-every? predicate set)

Returns #t if every element of set satisfies predicate, or #f otherwise. Note that this differs from the SRFI 1 analogue because it does not return an element of the set.

(set-map comparator proc set)

Applies proc to each element of set in arbitrary order and returns a newly allocated set, created as if by (set comparator), which contains the results of the applications. For example:

```
(set-map string-ci-comparator symbol->string (set eq? 'foo 'bar 'baz))
     => (set string-ci-comparator "foo" "bar" "baz")
```

Note that, when proc defines a mapping that is not 1:1, some of the mapped objects may be equivalent in the sense of comparator's equality predicate, and in this case duplicate elements are omitted as in the set constructor. For example:

```
(set-map (lambda (x) (quotient x 2))
         integer-comparator
         (set integer-comparator 1 2 3 4 5))
 => (set integer-comparator 0 1 2)
```

If the elements are the same in the sense of eqv?, it is unpredictable which one will be preserved in the result.

`(set-for-each proc set)`

Applies proc to set in arbitrary order, discarding the returned values. Returns an unspecified result.

`(set-fold proc nil set)`

Invokes proc on each member of set in arbitrary order, passing the result of the previous invocation as a second argument. For the first invocation, nil is used as the second argument. Returns the result of the last invocation, or nil if there was no invocation.

`(set-filter predicate set)`

Returns a newly allocated set with the same comparator as set, containing just the elements of set that satisfy predicate.

`(set-filter! predicate set)`

A linear update procedure that returns a set containing just the elements of set that satisfy predicate.

`(set-remove predicate set)`

Returns a newly allocated set with the same comparator as set, containing just the elements of set that do not satisfy predicate.

`(set-remove! predicate set)`

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

`(set-partition predicate set)`

Returns two values: a newly allocated set with the same comparator as set that contains just the elements of set that satisfy predicate, and another newly allocated set, also with the same comparator, that contains just the elements of set that do not satisfy predicate.

`(set-partition! predicate set)`

A linear update procedure that returns two sets containing the elements of set that do and do not, respectively, not satisfy predicate.

`(set-copy set)`

Returns a newly allocated set containing the elements of set, and using the same comparator.

`(set->list set)`

Returns a newly allocated list containing the members of set in unspecified order.

`(list->set comparator list)`

Returns a newly allocated set, created as if by set using comparator, that contains the elements of list. Duplicate elements (in the sense of the equality predicate) are omitted.

`(list->set! set list)`

Returns a set that contains the elements of both set and list. Duplicate elements (in the sense of the equality predicate) are omitted.

`(set=? set1 set2 ...)`

Returns #t if each set contains the same elements.

`(set<? set1 set2 ...)`

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

`(set>? set1 set2 ...)`

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

`(set<=? set1 set2 ...)`

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

`(set>=? set1 set2 ...)`

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

`(set-union set1 set2 ...)`

`(set-intersection set1 set2 ...)`

`(set-difference set1 set2 ...)`

`(set-xor set1 set2)`

Return a newly allocated set that is the union, intersection, asymmetric difference, or symmetric difference of the sets. Asymmetric difference is extended to more than two sets by taking the difference between the first set and the union of the others. Symmetric difference is not extended beyond two sets. Elements in the result set are drawn from the first set in which they appear.

`(set-union! set1 set2 ...)`

`(set-intersection! set1 set2 ...)`

`(set-difference! set1 set2 ...)`

`(set-xor! set1 set2)`

Linear update procedures returning a set that is the union, intersection, asymmetric difference, or symmetric difference of the sets. Asymmetric difference is extended to more than two sets by taking the difference between the first set and the union of the others. Symmetric difference is not extended beyond two sets. Elements in the result set are drawn from the first set in which they appear.

`(bag comparator element ...)`

`(bag-unfold ...)`

`(bag? obj)`

`(bag-contains? ...)`

(bag-empty? obj)‘

(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-sum set1 set2 ...)

(bag-sum! bag1 bag2 ...)

The bag-sum procedure returns a newly allocated bag containing all the unique elements in all the bags, such that the count of each unique element in the result is equal to the sum of the counts of that element in the arguments. It differs from bag-union by treating identical elements as potentially distinct rather than attempting to match them up.

The bag-sum! procedure is equivalent except that it is linear-update.

(bag-product n bag)

(bag-product! n bag)

The bag-product procedure returns a newly allocated bag containing all the unique elements in bag, where the count of each unique element in the bag is equal to the count of that element in bag multiplied by n.

The bag-product! procedure is equivalent except that it is linear-update.

(bag-unique-size bag)

Returns the number of unique elements of bag.

(bag-element-count bag element)

Returns an exact integer representing the number of times that element appears in bag.

(bag-for-each-unique proc bag)

Applies proc to each unique element of bag in arbitrary order, passing the element and the number of times it occurs in bag, and discarding the returned values. Returns an unspecified result.

(bag-fold-unique proc nil bag)

Invokes proc on each unique element of bag in arbitrary order, passing the number of occurrences as a second argument and the result of the previous invocation as a third argument. For the first invocation, nil is used as the third argument. Returns the result of the last invocation.

(bag-increment! bag element count)

(bag-decrement! bag element count)

Linear update procedures that return a bag with the same elements as bag, but with the element count of element in bag increased or decreased by the exact integer count (but not less than zero).

(bag->set bag)

(set->bag set)

(set->bag! bag set)

The bag->set procedure returns a newly allocated set containing the unique elements (in the sense of the equality predicate) of bag. The set->bag procedure returns a newly allocated bag containing the elements of set. The set->bag! procedure returns a bag containing the elements of both bag and set. In all cases, the comparator of the result is the same as the comparator of the argument or arguments.

(bag->alist bag)

(alist->bag comparator alist)

The bag->alist procedure returns a newly allocated alist whose keys are the unique elements of bag and whose values are the number of occurrences of each element. The alist->bag returning a newly allocated bag based on comparator, where the keys of alist specify the elements and the corresponding values of alist specify how many times they occur. Comparators

set-comparator

bag-comparator

Note that these comparators do not provide comparison procedures, as there is no ordering between sets or bags. It is an error to compare sets or bags with different element comparators. # `(scheme complex)`

angle

TODO

imag-part

TODO

magnitude

TODO

make-polar

TODO

make-rectangular

TODO

real-part

TODO # `(scheme division)`

This is based on SRFI-141.

This SRFI provides a fairly complete set of integral division and remainder operators.

(floor/ numerator denominator)

(floor-quotient numerator denominator)

(floor-remainder numerator denominator)

q = floor(n/d)

Thus r is negative iff d is negative.

(ceiling/ numerator denominator)

(ceiling-quotient numerator denominator)

(ceiling-remainder numerator denominator)

q = ceiling(n/d)

Thus r is negative iff d is non-negative.

If denominator is the number of units in a block, and is some number of units, then (ceiling-quotient numerator denominator) gives the number of blocks needed to cover numerator units. For example, denominator might be the number of bytes in a disk sector, and numerator the number of bytes in a file; then the quotient is the number of disk sectors needed to store the contents of the file. For another example, denominator might be the number of octets in the output of a cryptographic hash function, and numerator the number of octets desired in a key for a symmetric cipher, to be derived using the cryptographic hash function; then the quotient is the number of hash values needed to concatenate to make a key.

(truncate/ numerator denominator)

(truncate-quotient numerator denominator)

(truncate-remainder numerator denominator)

q = truncate(n/d)

Thus r is negative iff n is negative. However, by any non-unit denominator, the quotient of +1, 0, or -1 is 0; that is, three contiguous numerators by a common denominator share a common quotient. Of the other division operator pairs, only the round pair exhibits this property.

(round/ numerator denominator)

(round-quotient numerator denominator)

(round-remainder numerator denominator)

q = round(n/d)

The round function rounds to the nearest integer, breaking ties by choosing the nearest even integer. Nothing general can be said about the sign of r. Like the truncate operator pair, the quotient of +1, 0, or -1 by any non-unit denominator is 0, so that three contiguous numerators by a common denominator share a common quotient.

(euclidean/ numerator denominator)

(euclidean-quotient numerator denominator)

(euclidean-remainder numerator denominator)

If d > 0, q = floor(n/d); if d < 0, q = ceiling(n/d).

This division operator pair satisfies the stronger property

0 <= r < |d|,

used often in mathematics. Thus, for example, (euclidean-remainder numerator denominator) is always a valid index into a vector whose length is at least the absolute value of denominator. This division operator pair is so named because it is the subject of the Euclidean division algorithm.

(balanced/ numerator denominator)

(balanced-quotient numerator denominator)

(balanced-remainder numerator denominator)

This division operator pair satisfies the property

-|d/2| <= r < |d/2|.

When d is a power of 2, say 2k for some k, this reduces to

-2(k - 1) <= r < 2(k - 1).

Computer scientists will immediately recognize this as the interval of integers representable in two's-complement with k bits. # `(scheme bytevector)`

This is based on R6RS bytevectors library

(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 `<endianness symbol>`, 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.

(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.

(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)

(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)

(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)
(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)
(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)-

(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 generator)

This is based on SRFI-158

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.

(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)
```

`(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)`

qReturns 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.

(scheme flonum)

This is based on SRFI-144.

This library describes numeric procedures applicable to flonums, a subset of the inexact real numbers provided by a Scheme implementation. In most Schemes, the flonums and the inexact reals are the same. These procedures are semantically equivalent to the corresponding generic procedures, but allow more efficient implementations.

fl-e

Bound to the mathematical constant e. (C99 M_E)

fl-1/e

Bound to 1/e. (C99 M_E)

fl-e-2

Bound to e2.

fl-e-pi/4

Bound to e /4.

fl-log2-e

Bound to log2 e. (C99 M_LOG2E)

fl-log10-e

Bound to log10 e. (C99 M_LOG10E)

fl-log-2

Bound to loge 2. (C99 M_LN2)

fl-1/log-2

Bound to 1/loge 2. (C99 M_LN2)

fl-log-3

Bound to loge 3.

fl-log-pi

Bound to loge .

fl-log-10

Bound to loge 10. (C99 M_LN10)

fl-1/log-10

Bound to 1/loge 10. (C99 M_LN10)

fl-pi

Bound to the mathematical constant . (C99 M_PI)

fl-1/pi

Bound to 1/ . (C99 M_1_PI)

fl-2pi

Bound to 2 .

fl-pi/2

Bound to /2. (C99 M_PI_2)

fl-pi/4

Bound to /4. (C99 M_PI_4)

fl-pi-squared

Bound to 2.

fl-degree

Bound to /180, the number of radians in a degree.

fl-2/pi

Bound to 2/ . (C99 M_2_PI)

fl-2/sqrt-pi

Bound to 2/√ . (C99 M_2_SQRTPI)

fl-sqrt-2

Bound to √2. (C99 M_SQRT2)

fl-sqrt-3

Bound to √3.

fl-sqrt-5

Bound to √5.

fl-sqrt-10

Bound to √10.

fl-1/sqrt-2

Bound to 1/√2. (C99 M_SQRT1_2)

fl-cbrt-2

Bound to 2.

fl-cbrt-3

Bound to 3.

fl-4thrt-2

Bound to 2.

fl-phi‘

Bound to the mathematical constant .

fl-log-phi

Bound to log( ).

fl-1/log-phi

Bound to 1/log( ).

fl-euler

Bound to the mathematical constant (Euler's constant).

fl-e-euler

Bound to e .

fl-sin-1

Bound to sin 1.

fl-cos-1

Bound to cos 1.

fl-gamma-1/2

Bound to $\Gamma(1/2)$.

fl-gamma-1/3

Bound to $\Gamma(1/3)$.

fl-gamma-2/3

Bound to $\Gamma(2/3)$.

fl-greatest

fl-least

Bound to the largest/smallest positive finite flonum. (e.g. C99 DBL_MAX and C11 DBL_TRUE_MIN)

fl-epsilon

Bound to the appropriate machine epsilon for the hardware representation of flonums. (C99 DBL_EPSILON in <float.h>)

fl-fast-fl+*

Bound to #t if (fl+* x y z) executes about as fast as, or faster than, (fl+ (fl* x y) z); bound to #f otherwise. (C99 FP_FAST_FMA)

So that the value of this variable can be determined at compile time, R7RS implementations and other implementations that provide a features function should provide the feature fl-fast-fl+* if this variable is true, and not if it is false or the value is unknown at compile time.

fl-integer-exponent-zero

Bound to whatever exact integer is returned by (flinteger-exponent 0.0). (C99 FP_ILOGB0)

fl-integer-exponent-nan

Bound to whatever exact integer is returned by (flinteger-exponent +nan.0). (C99 FP_ILOGBNAN)

(flonum number)

If number is an inexact real number and there exists a flonum that is the same (in the sense of =) to number, returns that flonum. If number is a negative zero, an infinity, or a NaN, return its flonum equivalent. If such a flonum does not exist, returns the nearest flonum, where "nearest" is implementation-dependent. If number is not a real number, it is an error. If number is exact, applies inexact or exact->inexact to number first.

(fladjacent x y)

Returns a flonum adjacent to x in the direction of y. Specifically: if $x < y$, returns the smallest flonum larger than x; if $x > y$, returns the largest flonum smaller than x; if $x = y$, returns x. (C99 nextafter)

(flcopysign x y)

Returns a flonum whose magnitude is the magnitude of x and whose sign is the sign of y. (C99 copysign)

(make-flonum x n)

Returns $x \times 2n$, where n is an integer with an implementation-dependent range. (C99 ldexp)

(flinteger-fraction x)

Returns two values, the integral part of x as a flonum and the fractional part of x as a flonum. (C99 modf)

(flexponent x)

Returns the exponent of x. (C99 logb)

(flinteger-exponent x)

Returns the same as flexponent truncated to an exact integer. If x is zero, returns fl-integer-exponent-zero; if x is a NaN, returns fl-integer-exponent-nan; if x is infinite, returns a large implementation-dependent exact integer. (C99 ilogb)

(flnormalized-fraction-exponent x)

Returns two values, a correctly signed fraction y whose absolute value is between 0.5 (inclusive) and 1.0 (exclusive), and an exact integer exponent n such that x = y(2n). (C99 frexp)

(flsign-bit x)

Returns 0 for positive flonums and 1 for negative flonums and -0.0. The value of (flsign-bit +nan.0) is implementation-dependent, reflecting the sign bit of the underlying representation of NaNs. (C99 signbit)

(flonum? obj)

Returns #t if obj is a flonum and #f otherwise.

(fl=? x y z ...)

(fl<? x y z ...)

(fl>? x y z ...)

(fl<=? x y z ...)

(fl>=? x y z ...)

These procedures return #t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; they return #f otherwise. These predicates must be transitive. (C99 =, <, > <=, >= operators respectively)

(flunordered? x y)

Returns #t if x and y are unordered according to IEEE rules. This means that one of them is a NaN.

These numerical predicates test a flonum for a particular property, returning #t or #f.

(flinteger? x)

Tests whether x is an integral flonum.

`(flzero? x)`

Tests whether x is zero. Beware of roundoff errors.

`(flpositive? x)`

Tests whether x is positive.

`(flnegative? x)`

Tests whether x is negative. Note that (flnegative? -0.0) must return #f; otherwise it would lose the correspondence with (fl<? -0.0 0.0), which is #f according to IEEE 754.

`(flodd? x)`

Tests whether the flonum x is odd. It is an error if x is not an integer.

`(fleven? x)`

Tests whether the flonum x is even. It is an error if x is not an integer.

`(flfinite? x)`

Tests whether the flonum x is finite. (C99 isfinite)

`(flinfinite? x)`

Tests whether the flonum x is infinite. (C99 isinf)

`(flnan? x)`

Tests whether the flonum x is NaN. (C99 isnan)

`(flnormalized? x)`

Tests whether the flonum x is normalized. (C11 isnormal; in C99, use fpclassify(x) == FP_NORMAL)

`(fldenormalized? x)`

Tests whether the flonum x is denormalized. (C11 issubnormal; in C99, use fpclassify(x) == FP_SUBNORMAL)

(flmax x ...)

(flmin x ...)

Return the maximum/minimum argument. If there are no arguments, these procedures return -inf.0 or +inf.0 if the implementation provides these numbers, and (fl- fl-greatest) or fl-greatest otherwise. (C99 fmax fmin)

(fl+ x ...)

(fl* x ...)

Return the flonum sum or product of their flonum arguments. (C99 + * operators respectively)

(fl+* x y z)

Returns xy + z as if to infinite precision and rounded only once. The boolean constant fl-fast-fl+* indicates whether this procedure executes about as fast as, or faster than, a multiply and an add of flonums. (C99 fma)

(fl- x y ...)

(fl/ x y ...)

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument. (C99 - / operators respectively)

(flabs x)

Returns the absolute value of x. (C99 fabs)

(flabsdiff x y)

Returns |x - y|.

(flposdiff x y)

Returns the difference of x and y if it is non-negative, or zero if the difference is negative. (C99 fdim)

(flsgn x)

Returns (flcopysign 1.0 x).

(flnumerator x)

(fldenominator x)

Returns the numerator/denominator of x as a flonum; the result is computed as if x was represented as a fraction in lowest terms. The denominator is always positive. The numerator of an infinite flonum is itself. The denominator of an infinite or zero flonum is 1.0. The numerator and denominator of a NaN is a NaN.

(flfloor x)

Returns the largest integral flonum not larger than x. (C99 floor)

(flceiling x)

Returns the smallest integral flonum not smaller than x. (C99 ceil)

(flround x)

Returns the closest integral flonum to x, rounding to even when x represents a number halfway between two integers. (Not the same as C99 round, which rounds away from zero)

(fltruncate x)

Returns the closest integral flonum to x whose absolute value is not larger than the absolute value of x (C99 trunc) Exponents and logarithms

(flexp x)

Returns ex. (C99 exp)

(flexp2 x)

Returns 2x. (C99 exp2)

(flexp-1 x)

Returns ex - 1, but is much more accurate than flexp for very small values of x. It is recommended for use in algorithms where accuracy is important. (C99 expm1)

(flsquare x)

Returns x2.

(flsqrt x)

Returns √x. For -0.0, flsqrt should return -0.0. (C99 sqrt)

(flcbrt x)

Returns x. (C99 cbrt)

(flhypot x y)

Returns the length of the hypotenuse of a right triangle whose sides are of length |x| and |y|. (C99 hypot)

(flexpt x y)

Returns xy. If x is zero, then the result is zero. (C99 pow)

(fllog x)

Returns loge x. (C99 log)

(fllog1+ x)

Returns loge (x+ 1), but is much more accurate than fllog for values of x near 0. It is recommended for use in algorithms where accuracy is important. (C99 log1p)

(fllog2 x)

Returns log2 x. (C99 log2)

(fllog10 x)

Returns log10 x. (C99 log10)

(make-fllog-base x)

Returns a procedure that calculates the base-x logarithm of its argument. If x is 1.0 or less than 1.0, it is an error.

(flsin x)

Returns sin x. (C99 sin)

(flcos x)

Returns cos x. (C99 cos)

(fltan x)

Returns tan x. (C99 tan)

(flasin x)

Returns arcsin x. (C99 asin)

(flacos x)

Returns arccos x. (C99 acos)

(flatan [y] x)

Returns arctan x. (C99 atan)

With two arguments, returns arctan(y/x). in the range [- ,], using the signs of x and y to choose the correct quadrant for the result. (C99 atan2)

(flsinh x)

Returns sinh x. (C99 sinh)

(flcosh x)

Returns cosh x. (C99 cosh)

(fltanh x)

Returns tanh x. (C99 tanh)

(flasinh x)

Returns arcsinh x. (C99 asinh)

(flacosh x)

Returns arccosh x. (C99 acosh)

(flatanh x)

Returns arctanh x. (C99 atanh)

(flquotient x y)

Returns the quotient of x/y as an integral flonum, truncated towards zero.

`(flremainder x y)`

Returns the truncating remainder of x/y as an integral flonum.

`(flremquo x y)`

‘ Returns two values, the rounded remainder of x/y and the low-order n bits (as a correctly signed exact integer) of the rounded quotient. The value of n is implementation-dependent but at least 3. This procedure can be used to reduce the argument of the inverse trigonometric functions, while preserving the correct quadrant or octant. (C99 remquo)

`(flgamma x)`

Returns Γ(x), the gamma function applied to x. This is equal to (x-1)! for integers. (C99 tgamma)

`(flloggamma x)`

Returns two values, log $|\Gamma(x)|$ without internal overflow, and the sign of Γ(x) as 1.0 if it is positive and -1.0 if it is negative. (C99 lgamma)

`(flfirst-bessel n x)`

Returns the nth order Bessel function of the first kind applied to x, Jn(x). (jn, which is an XSI Extension of C99)

`(flsecond-bessel n x)`

Returns the nth order Bessel function of the second kind applied to x, Yn(x). (yn, which is an XSI Extension of C99)

`(flerf x)`

Returns the error function erf(x). (C99 erf)

`(flerfc x)`

Returns the complementary error function, 1 - erf(x). (C99 erfc) # `(scheme charset)`

This library is based on SRFI-14.

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.

(char-set? obj)

Is the object obj a character set?

(char-set= cs1 ...)

Are the character sets equal?

Boundary cases:

```
(char-set=) => true
(char-set= cs) => true
```

Rationale: transitive binary relations are generally extended to n-ary relations in Scheme, which enables clearer, more concise code to be written. While the zero-argument and one-argument cases will almost certainly not arise in first-order uses of such relations, they may well arise in higher-order cases or macro-generated code. E.g., consider

```
(apply char-set= cset-list)
```

This is well-defined if the list is empty or a singleton list. Hence we extend these relations to any number of arguments. Implementors have reported actual uses of n-ary relations in higher-order cases allowing for fewer than two arguments. The way of Scheme is to handle the general case; we provide the fully general extension.

A counter-argument to this extension is that R5RS's transitive binary arithmetic relations (=, <, etc.) require at least two arguments, hence this decision is a break with the prior convention – although it is at least one that is backwards-compatible.

(char-set<= cs1 ...)

Returns true if every character set csi is a subset of character set csi+1.

Boundary cases:

```
(char-set<=) => true
(char-set<= cs) => true
```

Rationale: See char-set= for discussion of zero- and one-argument applications. Consider testing a list of char-sets for monotonicity with

```
(apply char-set<= cset-list)
```

(char-set-hash cs [bound])

Compute a hash value for the character set cs. Bound is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

If bound is either zero or not given, the implementation may use an implementation-specific default value, chosen to be as large as is efficiently practical. For instance, the default range might be chosen for a given implementation to map all strings into the range of integers that can be represented with a single machine word.

Invariant:

```
(char-set= cs1 cs2) => (= (char-set-hash cs1 b) (char-set-hash cs2 b))
```

A legal but nonetheless discouraged implementation:

```
(define (char-set-hash cs . maybe-bound) 1)
```

Rationale: allowing the user to specify an explicit bound simplifies user code by removing the mod operation that typically accompanies every hash computation, and also may allow the implementation of the hash function to exploit a reduced range to efficiently compute the hash value. E.g., for small bounds, the hash function may be computed in a fashion such that intermediate values never overflow into bignum integers, allowing the implementor to provide a fixnum-specific "fast path" for computing the common cases very rapidly.

(char-set-cursor cset)

(char-set-ref cset cursor)

(char-set-cursor-next cset cursor)

(end-of-char-set? cursor)

Cursors are a low-level facility for iterating over the characters in a set. A cursor is a value that indexes a character in a char set. char-set-cursor produces a new cursor for a given char set. The set element indexed by the cursor is fetched with char-set-ref. A cursor index is incremented with char-set-cursor-next; in this way, code can step through every character in a char set. Stepping a cursor "past the end" of a char set produces a cursor that answers true to end-of-char-set?. It is an error to pass such a cursor to char-set-ref or to char-set-cursor-next.

A cursor value may not be used in conjunction with a different character set; if it is passed to char-set-ref or char-set-cursor-next with a character set other than the one used to create it, the results and effects are undefined.

Cursor values are not necessarily distinct from other types. They may be integers, linked lists, records, procedures or other values. This license is granted to allow cursors to be very "lightweight" values suitable for tight iteration, even in fairly simple implementations.

Note that these primitives are necessary to export an iteration facility for char sets to loop macros.

Example:

```
(define cs (char-set #\G #\a #\T #\e #\c #\h))

;; Collect elts of CS into a list.
(let lp ((cur (char-set-cursor cs)) (ans '()))
  (if (end-of-char-set? cur) ans
      (lp (char-set-cursor-next cs cur)
          (cons (char-set-ref cs cur) ans))))
  => (#\G #\T #\a #\c #\e #\h)

;; Equivalently, using a list unfold (from SRFI 1):
(unfold-right end-of-char-set?
              (curry char-set-ref cs)
              (curry char-set-cursor-next cs)
              (char-set-cursor cs))
  => (#\G #\T #\a #\c #\e #\h)
```

Rationale: Note that the cursor API's four functions "fit" the functional protocol used by the unfolders provided by the list, string and char-set SRFIs (see the example above). By way of contrast, here is a simpler, two-function API that was rejected for failing this criterion. Besides char-set-cursor, it provided a single function that mapped a cursor and a character set to two values, the indexed character and the next cursor. If the cursor had exhausted the character set, then this function returned false instead of the character value, and another end-of-char-set cursor. In this way, the other three functions of the current API were combined together.

(char-set-fold kons knil cs)

This is the fundamental iterator for character sets. Applies the function kons across the character set cs using initial state value knil. That is, if cs is the empty set, the procedure returns knil. Otherwise, some element c of cs is chosen; let cs' be the remaining, unchosen characters. The procedure returns

```
(char-set-fold kons (kons c knil) cs')

Examples:

;; CHAR-SET-MEMBERS
(lambda (cs) (char-set-fold cons '() cs))

;; CHAR-SET-SIZE
(lambda (cs) (char-set-fold (lambda (c i) (+ i 1)) 0 cs))

;; How many vowels in the char set?
(lambda (cs)
  (char-set-fold (lambda (c i) (if (vowel? c) (+ i 1) i))
                 0 cs))
```

(char-set-unfold f p g seed [base-cs])

(char-set-unfold! f p g seed base-cs)

This is a fundamental constructor for char-sets.

- G is used to generate a series of "seed" values from the initial seed: seed, (g seed), (g2 seed), (g3 seed), …
- P tells us when to stop – when it returns true when applied to one of these seed values.
- F maps each seed value to a character. These characters are added to the base character set base-cs to form the result; base-cs defaults to the empty set. char-set-unfold! adds the characters to base-cs in a linear-update – it is allowed, but not required, to side-effect and use base-cs's storage to construct the result.

More precisely, the following definitions hold, ignoring the optional-argument issues:

```
(define (char-set-unfold p f g seed base-cs)
  (char-set-unfold! p f g seed (char-set-copy base-cs)))

(define (char-set-unfold! p f g seed base-cs)
  (let lp ((seed seed) (cs base-cs))
       (if (p seed) cs                                  ; P says we are done.
           (lp (g seed)                                 ; Loop on (G SEED).
               (char-set-adjoin! cs (f seed))))))       ; Add (F SEED) to set.
```

(Note that the actual implementation may be more efficient.)

Examples:

```
(port->char-set p) = (char-set-unfold eof-object? values
                                      (lambda (x) (read-char p))
                                      (read-char p))

(list->char-set lis) = (char-set-unfold null? car cdr lis)
```

(char-set-for-each proc cs)

Apply procedure proc to each character in the character set cs. Note that the order in which proc is applied to the characters in the set is not specified, and may even change from one procedure application to another.

Nothing at all is specified about the value returned by this procedure; it is not even required to be consistent from call to call. It is simply required to be a value (or values) that may be passed to a command continuation, e.g. as the value of an expression appearing as a non-terminal subform of a begin expression. Note

that in R5RS, this restricts the procedure to returning a single value; non-R5RS systems may not even provide this restriction. char-set-map proc cs -> char-set proc is a char->char procedure. Apply it to all the characters in the char-set cs, and collect the results into a new character set.

Essentially lifts proc from a char->char procedure to a char-set -> char-set procedure.

Example:

```
(char-set-map char-downcase cset)
```

(char-set-copy cs)

Returns a copy of the character set cs. "Copy" means that if either the input parameter or the result value of this procedure is passed to one of the linear-update procedures described below, the other character set is guaranteed not to be altered.

A system that provides pure-functional implementations of the linear-operator suite could implement this procedure as the identity function – so copies are not guaranteed to be distinct by eq?.

(char-set char1 ...)

Return a character set containing the given characters.

(list->char-set char-list [base-cs])

(list->char-set! char-list base-cs)

Return a character set containing the characters in the list of characters char-list.

If character set base-cs is provided, the characters from char-list are added to it. list->char-set! is allowed, but not required, to side-effect and reuse the storage in base-cs; list->char-set produces a fresh character set.

(string->char-set s [base-cs])

(string->char-set! s base-cs)

Return a character set containing the characters in the string s.

If character set base-cs is provided, the characters from s are added to it. string->char-set! is allowed, but not required, to side-effect and reuse the storage in base-cs; string->char-set produces a fresh character set.

(char-set-filter pred cs [base-cs])

(char-set-filter! pred cs base-cs)

Returns a character set containing every character c in cs such that (pred c) returns true.

If character set base-cs is provided, the characters specified by pred are added to it. char-set-filter! is allowed, but not required, to side-effect and reuse the storage in base-cs; char-set-filter produces a fresh character set.

An implementation may not save away a reference to pred and invoke it after char-set-filter or char-set-filter! returns – that is, "lazy," on-demand implementations are not allowed, as pred may have external dependencies on mutable data or have other side-effects.

Rationale: This procedure provides a means of converting a character predicate into its equivalent character set; the cs parameter allows the programmer to bound the predicate's domain. Programmers should be aware that filtering a character set such as char-set:full could be a very expensive operation in an implementation that provided an extremely large character type, such as 32-bit Unicode. An earlier draft of this library provided a simple predicate->char-set procedure, which was rejected in favor of char-set-filter for this reason.

(ucs-range->char-set lower upper [error? base-cs])

(ucs-range->char-set! lower upper error? base-cs)

Lower and upper are exact non-negative integers; lower <= upper.

Returns a character set containing every character whose ISO/IEC 10646 UCS-4 code lies in the half-open range [lower,upper).

If the requested range includes unassigned UCS values, these are silently ignored (the current UCS specification has "holes" in the space of assigned codes).

If the requested range includes "private" or "user space" codes, these are handled in an implementation-specific manner; however, a UCS- or Unicode-based Scheme implementation should pass them through transparently.

If any code from the requested range specifies a valid, assigned UCS character that has no corresponding representative in the implementation's character type, then (1) an error is raised if error? is true, and (2) the code is ignored if error? is false (the default). This might happen, for example, if the implementation uses ASCII characters, and the requested range includes non-ASCII characters.

If character set base-cs is provided, the characters specified by the range are added to it. ucs-range->char-set! is allowed, but not required, to side-effect and reuse the storage in base-cs; ucs-range->char-set produces a fresh character set.

Note that ASCII codes are a subset of the Latin-1 codes, which are in turn a subset of the 16-bit Unicode codes, which are themselves a subset of the 32-bit UCS-4 codes. We commit to a specific encoding in this routine, regardless of the underlying representation of characters, so that client code using this library will be portable. I.e., a conformant Scheme implementation may use EBCDIC or SHIFT-JIS to encode characters; it must simply map the UCS characters from the given range into the native representation when possible, and report errors when not possible.

`(->char-set x)`

Coerces x into a char-set. X may be a string, character or char-set. A string is converted to the set of its constituent characters; a character is converted to a singleton set; a char-set is returned as-is. This procedure is intended for use by other procedures that want to provide "user-friendly," wide-spectrum interfaces to their clients.

`(char-set-size cs)`

Returns the number of elements in character set cs.

`(char-set-count pred cs)`

Apply pred to the chars of character set cs, and return the number of chars that caused the predicate to return true.

`(char-set->list cs)`

This procedure returns a list of the members of character set cs. The order in which cs's characters appear in the list is not defined, and may be different from one call to another.

`(char-set->string cs)`

This procedure returns a string containing the members of character set cs. The order in which cs's characters appear in the string is not defined, and may be different from one call to another.

`(char-set-contains? cs char)`

This procedure tests char for membership in character set cs.

The MIT Scheme character-set package called this procedure char-set-member?, but the argument order isn't consistent with the name.

(char-set-every pred cs)

(char-set-any pred cs)

The char-set-every procedure returns true if predicate pred returns true of every character in the character set cs. Likewise, char-set-any applies pred to every character in character set cs, and returns the first true value it finds. If no character produces a true value, it returns false. The order in which these procedures sequence through the elements of cs is not specified.

Note that if you need to determine the actual character on which a predicate returns true, use char-set-any and arrange for the predicate to return the character parameter as its true value, e.g.

```
(char-set-any (lambda (c) (and (char-upper-case? c) c))
              cs)
```

(char-set-adjoin cs char1 ...)

(char-set-delete cs char1 ...)

Add/delete the chari characters to/from character set cs.

(char-set-adjoin! cs char1 ...)

(char-set-delete! cs char1 ...)

Linear-update variants. These procedures are allowed, but not required, to side-effect their first parameter.

(char-set-complement cs)

(char-set-union cs1 ...)

(char-set-intersection cs1 ...)

(char-set-difference cs1 cs2 ...)

(char-set-xor cs1 ...)

(char-set-diff+intersection cs1 cs2 ...)

These procedures implement set complement, union, intersection, difference, and exclusive-or for character sets. The union, intersection and xor operations are n-ary. The difference function is also n-ary, associates to the left (that is, it computes the difference between its first argument and the union of all the other arguments), and requires at least one argument.

Boundary cases:

```
(char-set-union) => char-set:empty
(char-set-intersection) => char-set:full
(char-set-xor) => char-set:empty
(char-set-difference cs) => cs
```

char-set-diff+intersection returns both the difference and the intersection of the arguments – it partitions its first parameter. It is equivalent to

```
(values (char-set-difference cs1 cs2 ...)
        (char-set-intersection cs1 (char-set-union cs2 ...)))
```

but can be implemented more efficiently.

Programmers should be aware that char-set-complement could potentially be a very expensive operation in Scheme implementations that provide a very large character type, such as 32-bit Unicode. If this is a possibility, sets can be complimented with respect to a smaller universe using char-set-difference.

`(char-set-complement! cs)`

`(char-set-union! cs1 cs2 ...)`

`(char-set-intersection! cs1 cs2 ...)`

`(char-set-difference! cs1 cs2 ...)`

`(char-set-xor! cs1 cs2 ...)`

`(char-set-diff+intersection! cs1 cs2 cs3 ...)`

These are linear-update variants of the set-algebra functions. They are allowed, but not required, to side-effect their first (required) parameter.

char-set-diff+intersection! is allowed to side-effect both of its two required parameters, cs1 and cs2.

`char-set:lower-case`

Lower-case letters

`char-set:upper-case`

Upper-case letters

`char-set:title-case`

Title-case letters

`char-set:letter`

Letters

`char-set:digit`

Digits

`char-set:letter+digit`

Letters and digits

`char-set:graphic`

Printing characters except spaces

`char-set:printing`

Printing characters including spaces

`char-set:whitespace`

Whitespace characters

`char-set:iso-control`

The ISO control characters

`char-set:punctuation`

Punctuation characters

`char-set:symbol`

Symbol characters

`char-set:hex-digit`

A hexadecimal digit: 0-9, A-F, a-f

`char-set:blank`

Blank characters – horizontal whitespace

`char-set:ascii`

All characters in the ASCII set.

char-set:empty

Empty set

char-set:full

All characters # `(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 .

(emergency-exit [obj])

Terminates the program without running any outstanding dynamic-wind after procedures and communicates an exit value to the operating sstem 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 oerating , 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 canâ€™t decode the value. It is also an error to mutate the resulting .

(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 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 text)

This is based on SRFI-135.

In Scheme, strings are a mutable data type. Although it “is an error” (R5RS and R7RS) to use string-set! on literal strings or on strings returned by symbol->string, and any attempt to do so “should raise an exception” (R6RS), all other strings are mutable.

Although many mutable strings are never actually mutated, the mere possibility of mutation complicates specifications of libraries that use strings, encourages precautionary copying of strings, and precludes structure sharing that could otherwise be used to make procedures such as substring and string-append faster and more space-efficient.

This SRFI specifies a new data type of immutable texts. It comes with efficient and portable sample implementations that guarantee O(1) indexing for both sequential and random access, even in systems whose string-ref procedure takes linear time.

The operations of this new data type include analogues for all of the non-mutating operations on strings specified by the R7RS and most of those specified by SRFI 130, but the immutability of texts and uniformity of character-based indexing simplify the specification of those operations while avoiding several inefficiencies associated with the mutability of Scheme's strings.

`(text? obj)`

Is obj an immutable text? In particular, (text? obj) returns false if (string? obj) returns true, which implies string? returns false if text? returns true. Must execute in O(1) time.

`(textual? obj)`

Returns true if and only obj is an immutable text or a string. Must execute in O(1) time.

`(textual-null? text)`

Is text the empty text? Must execute in O(1) time.

`(textual-every pred textual [start end])`

`(textual-any pred textual [start end])`

Checks to see if every/any character in textual satisfies pred, proceeding from left (index start) to right (index end). textual-every These procedures are short-circuiting: if pred returns false, textual-every does not call pred on subsequent characters; if pred returns true, textual-any does not call pred on subsequent characters; Both procedures are "witness-generating":

- If textual-every is given an empty interval (with start = end), it returns #t.
- If textual-every returns true for a non-empty interval (with start < end), the returned true value is the one returned by the final call to the predicate on (text-ref (textual-copy text) (- end 1)).
- If textual-any returns true, the returned true value is the one returned by the predicate.

Note: The names of these procedures do not end with a question mark. This indicates a general value is returned instead of a simple boolean (#t or #f).

`(make-text len char)`

Returns a text of the given length filled with the given character.

(text char ...)

Returns a text consisting of the given characters.

(text-tabulate proc len)

Proc is a procedure that accepts an exact integer as its argument and returns a character. Constructs a text of size len by calling proc on each value from 0 (inclusive) to len (exclusive) to produce the corresponding element of the text. The order in which proc is called on those indexes is not specified.

Rationale: Although text-unfold is more general, text-tabulate is likely to run faster for the common special case it implements.

(text-unfold stop? mapper successor seed [base make-final])

This is a fundamental constructor for texts.

- successor is used to generate a series of "seed" values from the initial seed:
- seed, (successor seed), (successor2 seed), (successor3 seed), ...
- stop? tells us when to stop — when it returns true when applied to one of these seed values.
- mapper maps each seed value to the corresponding character(s) in the result text, which are assembled into that text in left-to-right order. It is an error for mapper to return anything other than a character, string, or text.
- base is the optional initial/leftmost portion of the constructed text, which defaults to the empty text (text). It is an error if base is anything other than a character, string, or text.
- make-final is applied to the terminal seed value (on which stop? returns true) to produce the final/rightmost portion of the constructed text. It defaults to (lambda (x) (text)). It is an error for make-final to return anything other than a character, string, or text.

text-unfold is a fairly powerful text constructor. You can use it to convert a list to a text, read a port into a text, reverse a text, copy a text, and so forth. Examples:

```
(port->text p) = (text-unfold eof-object?
                              values
                              (lambda (x) (read-char p))
                              (read-char p))

(list->text lis) = (text-unfold null? car cdr lis)

(text-tabulate f size) = (text-unfold (lambda (i) (= i size)) f add1 0)
```

```
To map f over a list lis, producing a text:

(text-unfold null? (compose f car) cdr lis)
```

Interested functional programmers may enjoy noting that textual-fold-right and text-unfold are in some sense inverses. That is, given operations knull?, kar, kdr, kons, and knil satisfying

```
(kons (kar x) (kdr x)) = x  and  (knull? knil) = #t
```

then

```
(textual-fold-right kons knil (text-unfold knull? kar kdr x)) = x
```

and

```
(text-unfold knull? kar kdr (textual-fold-right kons knil text)) = text.
```

This combinator pattern is sometimes called an "anamorphism."

Note: Implementations should not allow the size of texts created by text-unfold to be limited by limits on stack size.

(text-unfold-right stop? mapper successor seed [base make-final])

This is a fundamental constructor for texts. It is the same as text-unfold except the results of mapper are assembled into the text in right-to-left order, base is the optional rightmost portion of the constructed text, and make-final produces the leftmost portion of the constructed text.

```
(text-unfold-right (lambda (n) (< n (char->integer #\A)))
                   (lambda (n) (char-downcase (integer->char n)))
                   (lambda (n) (- n 1))
                   (char->integer #\Z)
                   #\space
                   (lambda (n) " The English alphabet: "))
   => « The English alphabet: abcdefghijklmnopqrstuvwxyz »
```

(textual->text textual)

When given a text, textual->text just returns that text. When given a string, textual->text returns the result of calling string->text on that string. Signals an error when its argument is neither string nor text.

(textual->string textual [start end])

(textual->vector textual [start end])

(textual->list textual [start end])

textual->string, textual->vector, and textual->list return a newly allocated (unless empty) mutable string, vector, or list of the characters that make up the given subtext or substring.

(string->text string [start end])

(vector->text char-vector [start end])

(list->text char-list [start end])

These procedures return a text containing the characters of the given substring, subvector, or sublist. The behavior of the text will not be affected by subsequent mutation of the given string, vector, or list.

(reverse-list->text char-list)

An efficient implementation of (compose list->text reverse):

```
(reverse-list->text '(#\a #\B #\c)) → «cBa»
```

This is a common idiom in the epilogue of text-processing loops that accumulate their result using a list in reverse order. (See also textual-concatenate-reverse for the "chunked" variant.)

(textual->utf8 textual [start end])

(textual->utf16 textual [start end])

(textual->utf16be textual [start end])

(textual->utf16le textual [start end])

These procedures return a newly allocated (unless empty) bytevector containing a UTF-8 or UTF-16 encoding of the given subtext or substring.

The bytevectors returned by textual->utf8, textual->utf16be, and textual->utf16le do not contain a byte-order mark (BOM). textual->utf16be returns a big-endian encoding, while textual->utf16le returns a little-endian encoding.

The bytevectors returned by textual->utf16 begin with a BOM that declares an implementation-dependent endianness, and the bytevector elements following that BOM encode the given subtext or substring using that endianness.

Rationale: These procedures are consistent with the Unicode standard. Unicode suggests UTF-16 should default to big-endian, but Microsoft prefers little-endian.

(utf8->text bytevector [start end])

(utf16->text bytevector [start end])

(utf16be->text bytevector [start end])

(utf16le->text bytevector [start end])

These procedures interpret their bytevector argument as a UTF-8 or UTF-16 encoding of a sequence of characters, and return a text containing that sequence.

The bytevector subrange given to utf16->text may begin with a byte order mark (BOM); if so, that BOM determines whether the rest of the subrange is to be interpreted as big-endian or little-endian; in either case, the BOM will not become a character in the returned text. If the subrange does not begin with a BOM, it is decoded using the same implementation-dependent endianness used by textual->utf16.

The utf16be->text and utf16le->text procedures interpret their inputs as big-endian or little-endian, respectively. If a BOM is present, it is treated as a normal character and will become part of the result.

It is an error if the bytevector subrange given to utf8->text contains invalid UTF-8 byte sequences. For the other three procedures, it is an error if start or end are odd, or if the bytevector subrange contains invalid UTF-16 byte sequences.

(text-length text)

Returns the number of characters within the given text. Must execute in O(1) time.

(text-ref text idx)

Returns character text[idx], using 0-origin indexing. Must execute in O(1) time.

(textual-length textual)

(textual-ref textual idx)

textual-length returns the number of characters in textual, and textual-ref returns the character at character index idx, using 0-origin indexing. These procedures are the generalizations of text-length and text-ref to accept strings as well as texts. If textual is a text, they must execute in O(1) time, but there is no such requirement if textual is a string.

Rationale: These procedures may be more convenient than the text-only versions, but compilers may generate faster code for calls to the text-only versions.

(subtext text start end)

(subtextual textual start end)

These procedures return a text containing the characters of text or textual beginning with index start (inclusive) and ending with index end (exclusive).

If textual is a string, then that string does not share any storage with the result, so subsequent mutation of that string will not affect the text returned by subtextual. When the first argument is a text, as is required by subtext, implementations are encouraged to return a result that shares storage with that text, to whatever extent sharing is possible while maintaining some small fixed bound on the ratio of storage used by the shared representation divided by the storage that would be used by an unshared representation. In particular, these procedures should just return their first argument when that argument is a text, start is 0, and end is the length of that text.

(textual-copy textual [start end])

Returns a text containing the characters of textual beginning with index start (inclusive) and ending with index end (exclusive).

Unlike subtext and subtextual, the result of textual-copy never shares substructures that would retain characters or sequences of characters that are substructures of its first argument or previously allocated objects.

If textual-copy returns an empty text, that empty text may be eq? or eqv? to the text returned by (text). If the text returned by textual-copy is non-empty, then it is not eqv? to any previously extant object.

(textual-take textual nchars)

(textual-drop textual nchars)

(textual-take-right textual nchars)

(textual-drop-right textual nchars)

textual-take returns a text containing the first nchars of textual; textual-drop returns a text containing all but the first nchars of textual. textual-take-right returns a text containing the last nchars of textual; textual-drop-right returns a text containing all but the last nchars of textual.

If textual is a string, then that string does not share any storage with the result, so subsequent mutation of that string will not affect the text returned by these procedures. If textual is a text, implementations are encouraged to return a result that shares storage with that text (which is easily accomplished by using subtext to create the result).

```
(textual-take "Pete Szilagyi" 6) => «Pete S»
(textual-drop "Pete Szilagyi" 6) => «zilagyi»

(textual-take-right "Beta rules" 5) => «rules»
(textual-drop-right "Beta rules" 5) => «Beta »
```

It is an error to take or drop more characters than are in the text:

```
(textual-take "foo" 37) => error
```

(textual-pad textual len [char start end])

(textual-pad-right textual len [char start end])

Returns a text of length len comprised of the characters drawn from the given subrange of textual, padded on the left (right) by as many occurrences of the character char as needed. If textual has more than len chars, it is truncated on the left (right) to length len. char defaults to # .

If textual is a string, then that string does not share any storage with the result, so subsequent mutation of that string will not affect the text returned by these procedures. If textual is a text, implementations are encouraged to return a result that shares storage with that text whenever sharing would be space-efficient.

```
(textual-pad     "325" 5) => «  325»
(textual-pad   "71325" 5) => «71325»
(textual-pad "8871325" 5) => «71325»
```

(textual-trim textual [pred start end])

(textual-trim-right textual [pred start end])

(textual-trim-both textual [pred start end])

Returns a text obtained from the given subrange of textual by skipping over all characters on the left / on the right / on both sides that satisfy the second argument pred: pred defaults to char-whitespace?.

If textual is a string, then that string does not share any storage with the result, so subsequent mutation of that string will not affect the text returned by these procedures. If textual is a text, implementations are encouraged to return a result that shares storage with that text whenever sharing would be space-efficient.

```
(textual-trim-both "  The outlook wasn't brilliant,  \n\r")
=> «The outlook wasn't brilliant,»
```

(textual-replace textual1 textual2 start1 end1 [start2 end2])

Returns

```
(textual-append (subtextual textual1 0 start1)
                (subtextual textual2 start2 end2)
                (subtextual textual1 end1 (textual-length textual1)))
```

That is, the segment of characters in textual1 from start1 to end1 is replaced by the segment of characters in textual2 from start2 to end2. If start1=end1, this simply splices the characters drawn from textual2 into textual1 at that position.

Examples:

```
(textual-replace "The TCL programmer endured daily ridicule."
                 "another miserable perl drone" 4 7 8 22)
    => «The miserable perl programmer endured daily ridicule.»

(textual-replace "It's easy to code it up in Scheme." "lots of fun" 5 9)
    => «It's lots of fun to code it up in Scheme.»

(define (textual-insert s i t) (textual-replace s t i i))

(textual-insert "It's easy to code it up in Scheme." 5 "really ")
    => «It's really easy to code it up in Scheme.»

(define (textual-set s i c) (textual-replace s (text c) i (+ i 1)))

(textual-set "Text-ref runs in O(n) time." 19 #\1)
    => «Text-ref runs in O(1) time.»
```

(textual=? textual1 textual2 textual3 ...)

Returns #t if all the texts have the same length and contain exactly the same characters in the same positions; otherwise returns #f.

(textual<? textual1 textual2 textual3 ...)

(textual>? textual1 textual2 textual3 ...)

(textual<=? textual1 textual2 textual3 ...)

(textual>=? textual1 textual2 textual3 ...)

These procedures return #t if their arguments are (respectively): monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing.

These comparison predicates are required to be transitive.

These procedures compare texts in an implementation-defined way. One approach is to make them the lexicographic extensions to texts of the corresponding orderings on characters. In that case, text<? would be the lexicographic ordering on texts induced by the ordering char<? on characters, and if two texts differ in length but are the same up to the length of the shorter text, the shorter text would be considered to be lexicographically less than the longer string. However, implementations are also allowed to use more sophisticated locale-specific orderings.

In all cases, a pair of texts must satisfy exactly one of textual<?, textual=?, and textual>?, must satisfy textual<=? if and only if they do not satisfy textual>?, and must satisfy textual>=? if and only if they do not satisfy textual<?.

Note: Implementations are encouraged to use the same orderings for texts as are used by the corresponding comparisons on strings, but are allowed to use different orderings.

Rationale: The only portable way to ensure these comparison predicates use the same orderings used by the corresponding comparisons on strings is to convert all texts to strings, which would be unacceptably inefficient.

`(textual-ci=? textual1 textual2 textual3 ...)`

Returns #t if, after calling textual-foldcase on each of the arguments, all of the case-folded texts would have the same length and contain the same characters in the same positions; otherwise returns #f.

`(textual-ci<?  textual1 textual2 textual3 ...)`

`(textual-ci>?  textual1 textual2 textual3 ...)`

`(textual-ci<=? textual1 textual2 textual3 ...)`

`(textual-ci>=? textual1 textual2 textual3 ...)`

These procedures behave as though they had called textual-foldcase on their arguments before applying the corresponding procedures without "-ci".

`(textual-prefix-length textual1 textual2 [start1 end1 start2 end2])`

`(textual-suffix-length textual1 textual2 [start1 end1 start2 end2])`

Return the length of the longest common prefix/suffix of textual1 and textual2. For prefixes, this is equivalent to their "mismatch index" (relative to the start indexes).

The optional start/end indexes restrict the comparison to the indicated subtexts of textual1 and textual2.

(textual-prefix? textual1 textual2 [start1 end1 start2 end2])

(textual-suffix? textual1 textual2 [start1 end1 start2 end2])

Is textual1 a prefix/suffix of textual2?

The optional start/end indexes restrict the comparison to the indicated subtexts of textual1 and textual2.

(textual-index textual pred [start end])

(textual-index-right textual pred [start end])

(textual-skip textual pred [start end])

(textual-skip-right textual pred [start end])

textual-index searches through the given subtext or substring from the left, returning the index of the leftmost character satisfying the predicate pred. textual-index-right searches from the right, returning the index of the rightmost character satisfying the predicate pred. If no match is found, these procedures return #f.

Rationale: The SRFI 130 analogues of these procedures return cursors, even when no match is found, and SRFI 130's string-index-right returns the successor of the cursor for the first character that satisfies the predicate. As there are no cursors in this SRFI, it seems best to follow the more intuitive and long-standing precedent set by SRFI 13.

The start and end arguments specify the beginning and end of the search; the valid indexes relevant to the search include start but exclude end. Beware of "fencepost" errors: when searching right-to-left, the first index considered is (- end 1), whereas when searching left-to-right, the first index considered is start. That is, the start/end indexes describe the same half-open interval [start,end) in these procedures that they do in all other procedures specified by this SRFI.

The skip functions are similar, but use the complement of the criterion: they search for the first char that doesn't satisfy pred. To skip over initial whitespace, for example, say

```
(subtextual text
            (or (textual-skip text char-whitespace?)
                (textual-length text))
            (textual-length text))
```

These functions can be trivially composed with textual-take and textual-drop to produce take-while, drop-while, span, and break procedures without loss of efficiency.

(textual-contains textual1 textual2 [start1 end1 start2 end2])

(textual-contains-right textual1 textual2 [start1 end1 start2 end2])

Does the subtext of textual1 specified by start1 and end1 contain the sequence of characters given by the subtext of textual2 specified by start2 and end2?

Returns #f if there is no match. If start2 = end2, textual-contains returns start1 but textual-contains-right returns end1. Otherwise returns the index in textual1 for the first character of the first/last match; that index lies within the half-open interval [start1,end1), and the match lies entirely within the [start1,end1) range of textual1.

```
(textual-contains "eek -- what a geek." "ee" 12 18) ; Searches "a geek"
=> 15
```

Note: The names of these procedures do not end with a question mark. This indicates a useful value is returned when there is a match.

(textual-upcase textual)

(textual-downcase textual)

(textual-foldcase textual)

(textual-titlecase textual)

These procedures return the text obtained by applying Unicode's full uppercasing, lowercasing, case-folding, or title-casing algorithms to their argument. In some cases, the length of the result may be different from the length of the argument. Note that language-sensitive mappings and foldings are not used.

(textual-append textual ...)

Returns a text whose sequence of characters is the concatenation of the sequences of characters in the given arguments.

(textual-concatenate textual-list)

Concatenates the elements of textual-list together into a single text.

If any elements of textual-list are strings, then those strings do not share any storage with the result, so subsequent mutation of those string will not affect

the text returned by this procedure. Implementations are encouraged to return a result that shares storage with some of the texts in the list if that sharing would be space-efficient.

Rationale: Some implementations of Scheme limit the number of arguments that may be passed to an n-ary procedure, so the (apply textual-append textual-list) idiom, which is otherwise equivalent to using this procedure, is not as portable.

(textual-concatenate-reverse textual-list [final-textual end])

With no optional arguments, calling this procedure is equivalent to

```
(textual-concatenate (reverse textual-list))
```

If the optional argument final-textual is specified, it is effectively consed onto the beginning of textual-list before performing the list-reverse and textual-concatenate operations.

If the optional argument end is given, only the characters up to but not including end in final-textual are added to the result, thus producing

```
(textual-concatenate
  (reverse (cons (subtext final-textual 0 end)
                 textual-list)))
```

For example:

```
(textual-concatenate-reverse '(" must be" "Hello, I") "
    going.XXXX" 7) => «Hello, I must be going.» `` Rationale: This
    procedure is useful when constructing procedures that accumulate
    character data into lists of textual buffers, and wish to convert
    the accumulated data into a single text when done. The optional
    end argument accommodates that use case when final-textual is a
    mutable string, and is allowed (for uniformity) when final-textual
    is an immutable text.
```

(textual-join textual-list [delimiter grammar])

This procedure is a simple unparser; it pastes texts together using the delimiter text.

textual-list is a list of texts and/or strings. delimiter is a text or a string. The grammar argument is a symbol that determines how the delimiter is used, and defaults to 'infix. It is an error for grammar to be any symbol other than these four:

- 'infix means an infix or separator grammar: insert the delimiter between list elements. An empty list will produce an empty text.

- 'strict-infix means the same as 'infix if the textual-list is non-empty, but will signal an error if given an empty list. (This avoids an ambiguity shown in the examples below.)
- 'suffix means a suffix or terminator grammar: insert the delimiter after every list element.
- 'prefix means a prefix grammar: insert the delimiter before every list element.

The delimiter is the text used to delimit elements; it defaults to a single space " ".

```
(textual-join '("foo" "bar" "baz"))
         => «foo bar baz»
(textual-join '("foo" "bar" "baz") "")
         => «foobarbaz»
(textual-join '("foo" "bar" "baz") «:»)
         => «foo:bar:baz»
(textual-join '("foo" "bar" "baz") ":" 'suffix)
         => «foo:bar:baz:»

;; Infix grammar is ambiguous wrt empty list vs. empty text:
(textual-join '()   ":") => «»
(textual-join '("") ":") => «»

;; Suffix and prefix grammars are not:
(textual-join '()   ":" 'suffix)) => «»
(textual-join '("") ":" 'suffix)) => «:»
```

(textual-fold kons knil textual [start end])

(textual-fold-right kons knil textual [start end])

These are the fundamental iterators for texts.

The textual-fold procedure maps the kons procedure across the given text or string from left to right:

```
(... (kons textual[2] (kons textual[1] (kons textual[0] knil))))
```

In other words, textual-fold obeys the (tail) recursion

```
(textual-fold kons knil textual start end)
= (textual-fold kons (kons textual[start] knil) start+1 end)
```

The textual-fold-right procedure maps kons across the given text or string from right to left:

```
(kons textual[0]
      (... (kons textual[end-3]
```

```
                    (kons textual[end-2]
                          (kons textual[end-1]
                                knil)))))
```

obeying the (tail) recursion

```
  (textual-fold-right kons knil textual start end)
= (textual-fold-right kons (kons textual[end-1] knil) start end-1)
```

Examples:

```
;;; Convert a text or string to a list of chars.
(textual-fold-right cons '() textual)

;;; Count the number of lower-case characters in a text or string.
(textual-fold (lambda (c count)
                (if (char-lower-case? c)
                    (+ count 1)
                    count))
              0
              textual)
```

The textual-fold-right combinator is sometimes called a "catamorphism."

(textual-map proc textual1 textual2 ...)

It is an error if proc does not accept as many arguments as the number of textual arguments passed to textual-map, does not accept characters as arguments, or returns a value that is not a character, string, or text.

The textual-map procedure applies proc element-wise to the characters of the textual arguments, converts each value returned by proc to a text, and returns the concatenation of those texts. If more than one textual argument is given and not all have the same length, then textual-map terminates when the shortest textual argument runs out. The dynamic order in which proc is called on the characters of the textual arguments is unspecified, as is the dynamic order in which the coercions are performed. If any strings returned by proc are mutated after they have been returned and before the call to textual-map has returned, then textual-map returns a text with unspecified contents; the textual-map procedure itself does not mutate those strings.

Example:

```
(textual-map (lambda (c0 c1 c2)
               (case c0
                ((#\1) c1)
                ((#\2) (string c2))
                ((#\-) (text #\- c1))))
             (string->text "1222-1111-2222")
             (string->text "Hi There!")
```

```
            (string->text "Dear John"))
      => «Hear-here!»
```

(textual-for-each proc textual1 textual2 ...)

It is an error if proc does not accept as many arguments as the number of textual arguments passed to textual-map or does not accept characters as arguments.

The textual-for-each procedure applies proc element-wise to the characters of the textual arguments, going from left to right. If more than one textual argument is given and not all have the same length, then textual-for-each terminates when the shortest textual argument runs out.

(textual-map-index proc textual [start end])

Calls proc on each valid index of the specified subtext or substring, converts the results of those calls into texts, and returns the concatenation of those texts. It is an error for proc to return anything other than a character, string, or text. The dynamic order in which proc is called on the indexes is unspecified, as is the dynamic order in which the coercions are performed. If any strings returned by proc are mutated after they have been returned and before the call to textual-map-index has returned, then textual-map-index returns a text with unspecified contents; the textual-map-index procedure itself does not mutate those strings.

(textual-for-each-index proc textual [start end])

Calls proc on each valid index of the specified subtext or substring, in increasing order, discarding the results of those calls. This is simply a safe and correct way to loop over a subtext or substring.

Example:

```
(let ((txt (string->text "abcde"))
      (v '()))
  (textual-for-each-index
    (lambda (cur) (set! v (cons (char->integer (text-ref txt cur)) v)))
    txt)
  v) => (101 100 99 98 97)
```

(textual-count textual pred [start end])

Returns a count of the number of characters in the specified subtext of textual that satisfy the given predicate.

(textual-filter pred textual [start end])

(textual-remove pred textual [start end])

Filter the given subtext of textual, retaining only those characters that satisfy / do not satisfy pred.

If textual is a string, then that string does not share any storage with the result, so subsequent mutation of that string will not affect the text returned by these procedures. If textual is a text, implementations are encouraged to return a result that shares storage with that text whenever sharing would be space-efficient.

(textual-replicate textual from to [start end])

This is an "extended subtext" procedure that implements replicated copying of a subtext or substring.

textual is a text or string; start and end are optional arguments that specify a subtext of textual, defaulting to 0 and the length of textual. This subtext is conceptually replicated both up and down the index space, in both the positive and negative directions. For example, if textual is "abcdefg", start is 3, and end is6, then we have the conceptual bidirectionally-infinite text

```
... d  e  f  d  e  f  d  e  f  d  e  f  d  e  f  d  e  f  d ...
   -9 -8 -7 -6 -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5 +6 +7 +8 +9
```

textual-replicate returns the subtext of this text beginning at index from, and ending at to. It is an error if from is greater than to.

You can use textual-replicate to perform a variety of tasks:

- To rotate a text left: (textual-replicate "abcdef" 2 8) => «cdefab»
- To rotate a text right: (textual-replicate "abcdef" -2 4) => «efabcd»
- To replicate a text: (textual-replicate "abc" 0 7) => «abcabca»

Note that

- The from/to arguments give a half-open range containing the characters from index from up to, but not including, index to.
- The from/to indexes are not expressed in the index space of textual. They refer instead to the replicated index space of the subtext defined by textual, start, and end.

It is an error if start=end, unless from=to, which is allowed as a special case.

(textual-split textual delimiter [grammar limit start end])

Returns a list of texts representing the words contained in the subtext of textual from start (inclusive) to end (exclusive). The delimiter is a text or string to be

used as the word separator. This will often be a single character, but multiple characters are allowed for use cases such as splitting on "&". The returned list will have one more item than the number of non-overlapping occurrences of the delimiter in the text. If delimiter is an empty text, then the returned list contains a list of texts, each of which contains a single character.

The grammar is a symbol with the same meaning as in the textual-join procedure. If it is infix, which is the default, processing is done as described above, except an empty textual produces the empty list; if grammar is strict-infix, then an empty textual signals an error. The values prefix and suffix cause a leading/trailing empty text in the result to be suppressed.

If limit is a non-negative exact integer, at most that many splits occur, and the remainder of textual is returned as the final element of the list (so the result will have at most limit+1 elements). If limit is not specified or is #f, then as many splits as possible are made. It is an error if limit is any other value.

To split on a regular expression re, use SRFI 115's regexp-split procedure:

```
(map string->text (regexp-split re (textual->string txt)))
```

Rationale: Although it would be more efficient to have a version of regexp-split that operates on texts directly, the scope of this SRFI is limited to specifying operations on texts analogous to those specified for strings by R7RS and SRFI 130. # `(scheme idque)`

This is based on SRFI-134.

This SRFI defines immutable deques. A deque is a double-ended queue, a sequence which allows elements to be added or removed efficiently from either end. A structure is immutable when all its operations leave the structure unchanged. Note that none of the procedures specified here ends with an exclamation point.

This SRFI describes immutable deques, or ideques. Immutable structures are sometimes called persistent and are closely related to pure functional (a.k.a. pure) structures. The availability of immutable data structures facilitates writing efficient programs in the pure-functional style. Immutable deques can also be seen as a bidirectional generalization of immutable lists, and some of the procedures documented below are most useful in that context. Unlike the immutable lists of SRFI 116, it is efficient to produce modified versions of an ideque; unlike the list queues of SRFI 117, it is possible to efficiently return an updated version of an ideque without mutating any earlier versions of it.

The specification was designed jointly by Kevin Wortman and John Cowan. John Cowan is the editor and shepherd. The two-list implementation was written by John Cowan.

`(ideque element ...)`

Returns an ideque containing the elements. The first element (if any) will be at the front of the ideque and the last element (if any) will be at the back. Takes O(n) time, where n is the number of elements.

`(ideque-tabulate n proc)`

Invokes the predicate proc on every exact integer from 0 (inclusive) to n (exclusive). Returns an ideque containing the results in order of generation. Takes O(n) time.

`(ideque-unfold stop? mapper successor seed)`

Invokes the predicate stop? on seed. If it returns false, generate the next result by applying mapper to seed, generate the next seed by applying successor to seed, and repeat this algorithm with the new seed. If stop? returns true, return an ideque containing the results in order of accumulation. Takes O(n) time.

`(ideque-unfold-right stop? mapper successor seed)`

Invokes the predicate stop? on seed. If it returns false, generate the next result by applying mapper to seed, generate the next seed by applying successor to seed, and repeat the algorithm with the new seed. If stop? returns true, return an ideque containing the results in reverse order of accumulation. Takes O(n) time. Predicates

`(ideque? x)`

Returns #t if x is an ideque, and #f otherwise. Takes O(1) time.

`(ideque-empty? idaeque)`

Returns #t if ideque contains zero elements, and #f otherwise. Takes O(1) time.

`(ideque= elt= ideque ...)`

Determines ideque equality, given an element-equality procedure. Ideque A equals ideque B if they are of the same length, and their corresponding elements are equal, as determined by elt=. If the element-comparison procedure's first argument is from idequei, then its second argument is from idequei+1, i.e. it is always called as (elt= a b) for a an element of ideque A, and b an element of ideque B.

In the n-ary case, every idequei is compared to idequei+1 (as opposed, for example, to comparing ideque1 to every idequei, for $i > 1$). If there are zero or

one ideque arguments, ideque= simply returns true. The name does not end in a question mark for compatibility with the SRFI-1 procedure list=.

Note that the dynamic order in which the elt= procedure is applied to pairs of elements is not specified. For example, if ideque= is applied to three ideques, A, B, and C, it may first completely compare A to B, then compare B to C, or it may compare the first elements of A and B, then the first elements of B and C, then the second elements of A and B, and so forth.

The equality procedure must be consistent with eq?. Note that this implies that two ideques which are eq? are always ideque=, as well; implementations may exploit this fact to "short-cut" the element-by-element comparisons.

`(ideque-any pred ideque)`

`(ideque-every pred ideque)`

Invokes pred on the elements of the ideque in order until one call returns a true/false value, which is then returned. If there are no elements, returns #f/#t. Takes O(n) time. Queue operations

`(ideque-front ideque)`

`(ideque-back ideque)`

Returns the front/back element of ideque. It is an error for ideque to be empty. Takes O(1) time.

`(ideque-remove-front ideque)`

`(ideque-remove-back ideque)`

Returns an ideque with the front/back element of ideque removed. It is an error for ideque to be empty. Takes O(1) time.

`(ideque-add-front ideque obj)`

`(ideque-add-back ideque obj)`

Returns an ideque with obj pushed to the front/back of ideque. Takes O(1) time. Other accessors

`(ideque-ref ideque n)`

Returns the nth element of ideque. It is an error unless n is less than the length of ideque. Takes O(n) time.

(ideque-take ideque n)

(ideque-take-right ideque n)

Returns an ideque containing the first/last n elements of ideque. It is an error if n is greater than the length of ideque. Takes O(n) time.

(ideque-drop ideque n)

(ideque-drop-right ideque n)

Returns an ideque containing all but the first/last n elements of ideque. It is an error if n is greater than the length of ideque. Takes O(n) time.

(ideque-split-at ideque n)

Returns two values, the results of (ideque-take ideque n) and (ideque-drop ideque n) respectively, but may be more efficient. Takes O(n) time. The whole ideque

(ideque-length ideque)

Returns the length of ideque as an exact integer. May take O(n) time, though O(1) is optimal.

(ideque-append ideque ...)

Returns an ideque with the contents of the ideque followed by the others, or an empty ideque if there are none. Takes O(kn) time, where k is the number of ideques and n is the number of elements involved, though O(k log n) is possible.

(ideque-reverse ideque)

Returns an ideque containing the elements of ideque in reverse order. Takes O(1) time.

(ideque-count pred ideque)

Pred is a procedure taking a single value and returning a single value. It is applied element-wise to the elements of ideque, and a count is tallied of the number of elements that produce a true value. This count is returned. Takes O(n) time. The dynamic order of calls to pred is unspecified.

(ideque-zip ideque1 ideque2 ...)

Returns an ideque of lists (not ideques) each of which contains the corresponding elements of ideques in the order specified. Terminates when all the elements of

any of the ideques have been processed. Takes O(kn) time, where k is the number of ideques and n is the number of elements in the shortest ideque.

`(ideque-map proc ideque)`

Applies proc to the elements of ideque and returns an ideque containing the results in order. The dynamic order of calls to proc is unspecified. Takes O(n) time.

`(ideque-filter-map proc ideque)`

Applies proc to the elements of ideque and returns an ideque containing the true (i.e. non-#f) results in order. The dynamic order of calls to proc is unspecified. Takes O(n) time.

`(ideque-for-each proc ideque)`

`(ideque-for-each-right proc ideque)`

Applies proc to the elements of ideque in forward/reverse order and returns an unspecified result. Takes O(n) time.

`(ideque-fold proc nil ideque)`

`(ideque-fold-right proc nil ideque)`

Invokes proc on the elements of ideque in forward/reverse order, passing the result of the previous invocation as a second argument. For the first invocation, nil is used as the second argument. Returns the result of the last invocation, or nil if there was no invocation. Takes O(n) time.

`(ideque-append-map proc ideque)`

Applies proc to the elements of ideque. It is an error if the result is not a list. Returns an ideque containing the elements of the lists in order. Takes O(n) time, where n is the number of elements in all the lists returned.

`(ideque-filter pred ideque)`

`(ideque-remove pred ideque)`

Returns an ideque containing the elements of ideque that do/do not satisfy pred. Takes O(n) time.

`(ideque-partition proc ideque)`

Returns two values, the results of (ideque-filter pred ideque) and (ideque-remove pred ideque) respectively, but may be more efficient. Takes O(n) time.

(ideque-find pred ideque [failure])

(ideque-find-right pred ideque [failure])

Returns the first/last element of ideque that satisfies pred. If there is no such element, returns the result of invoking the thunk failure; the default thunk is (lambda () #f). Takes O(n) time.

(ideque-take-while pred ideque)

(ideque-take-while-right pred ideque)

Returns an ideque containing the longest initial/final prefix of elements in ideque all of which satisfy pred. Takes O(n) time.

(ideque-drop-while pred ideque)

(ideque-drop-while-right pred ideque)

Returns an ideque which omits the longest initial/final prefix of elements in ideque all of which satisfy pred, but includes all other elements of ideque. Takes O(n) time.

(ideque-span pred ideque)

(ideque-break pred ideque)

Returns two values, the initial prefix of the elements of ideque which do/do not satisfy pred, and the remaining elements. Takes O(n) time.

(list->ideque list)

(ideque->list ideque)

Conversion between ideque and list structures. FIFO order is preserved, so the front of a list corresponds to the front of an ideque. Each operation takes O(n) time.

(generator->ideque generator)

(ideque->generator ideque)

Conversion between SRFI 121 generators and ideques. Each operation takes O(n) time. A generator is a procedure that is called repeatedly with no arguments to generate consecutive values, and returns an end-of-file object when it has no more values to return. # `(scheme mapping)`

This library is based on SRFI-146.

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.

(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.

(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.

(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.

(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]])

(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)

(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.

`(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-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.

(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.

`(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.

(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.

(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.

`(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. # `(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 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 ...)`

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

`(< number number ...)`

Less than procedure. Return a boolean.

`(<= number number ...)`

Less than or equal procedure. Return a boolean.

`(= number number ...)`

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

`(> number number ...)`

Greater than procedure. Return a boolean.

`(>= number number ...)`

Greater than or equal. Return a boolean.

`(abs number)`

Return the absolute value of `NUMBER`.

`(and test1 ...)`

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

`(append lst ...)`

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

`(apply proc arg1 ... args)`

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

`(assoc obj alist)`

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

`(assq obj alist)`

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

`(assv obj alist)`

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

`begin` syntax

There 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`

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

`bytevector-u8-set!`

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`

TODO

`cadr`

TODO

`(call-with-current-continuation proc)`

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

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

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

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It can be stored in variables or data structures

and can be called as many times as desired. However, like the raise and error procedures, it never returns to its caller.

TODO: example

`(call-with-port port proc)`

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

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

`(call-with-values producer consumer)`

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

`(call/cc proc)`

Abbreviation for `call-with-continuation`.

`(car pair)`

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

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

TODO

`cdar`

TODO

`cddr`

TODO

`cdr`

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

`(ceiling x)`

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

`(char->integer char)`

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

`(char-ready? [port])`

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

`char<=?`

TODO

`char<?`

TODO

`char=?`

TODO

`char>=?`

TODO

`char>?`

TODO

`char?`

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

`(close-input-port port)`

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

`(close-output-port port)`

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

`(close-port port)`

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

`(complex? 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>)`

TODO

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

TODO

`define-record-type` syntax

TODO

`define-syntax`

TODO

`(define-values var1 ... expr)` syntax

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

`(denominator q)`

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

`do`

TODO

`(dynamic-wind before thunk after)`

TODO

`(eof-object)`

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

`(eof-object? obj)`

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

`(eq? obj1 obj2)`

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

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

`(equal? obj1 obj2)`

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

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

`(eqv? obj1 obj2)`

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

TODO: complete based on r7rs small and guile.

`(error [who] message . irritants)`

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

`(error-object-irritants error)`

Returns a list of the irritants encapsulated by error.

`(error-object-message error)`

Returns the message encapsulated by error.

`(error-object? obj)`

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

`(even? number)`

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

`(exact z)`

TODO: FIXME

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

`(exact-integer-sqrt k)`

TODO

`(exact-integer? z)`

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

`(exact? z)`

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

`(expt z1 z2)`

Returns `z1` raised to the power `z2`.

`features`

TODO: no implemented

`(file-error? error)`

TODO: not implemented?

`(floor x)`

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

`floor-quotient`

TODO

`floor-remainder`

TODO

`floor/`

TODO

`(flush-output-port [port])`

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

`(for-each proc list1 ...)`

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

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

`(gcd n1 ...)`

Return the greatest common divisor.

`(get-output-bytevector 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

TODO

let*

TODO

let*-values

TODO

let-syntax

TODO

let-values

TODO

letrec

TODO

letrec*

TODO

letrec-syntax

TODO

(list obj ...)

Returns a newly allocated list of its arguments.

(list->string list)

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

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

(list->vector list)

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

(list-copy obj)

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

(list-ref list k)

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

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

(list-set! list k obj)

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

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

(list-tail list k)

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

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

(list? obj)

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

(make-bytevector k [byte])

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

(make-list k [fill])

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

(make-parameter init [converter])

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

(make-string k [char])

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

(make-vector k [fill])

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

(map proc list1 ...)

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

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

(max x1 ...)

Return the maximum of its arguments.

(member obj list [compare])

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

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

`(memq obj list)`

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

Use `eq?` for comparison.

`(memv obj list)`

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

Uses `eqv?` for comparison.

`(min x1 ...)`

Return the minimum of its arguments.

`(modulo n1 n2)`

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

`(negative? x)`

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

`(newline [port])`

Writes an end of line to output port.

`(not obj)`

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

`(null? obj)`

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

`(number->string z [radix])`

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

`(number? obj)`

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

`(numerator q)`

TODO

`(odd? number)`

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

`(open-input-bytevector bytevector)`

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

`(open-input-string string)`

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

`(open-output-bytevector)`

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

`(open-output-string)`

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

`(or test1 ...)` syntax

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

`(output-port-open? port)`

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

`(output-port? obj)`

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

`(pair? obj)`

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

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

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

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

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

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

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

`(peek-char [port])`

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

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

`(peek-u8 [port])`

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

`(port? obj)`

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

`(positive? x)`

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

`(procedure? obj)`

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

`quasiquote`

TODO

`quote`

TODO

`quotient`

TODO

`(raise obj)`

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

`(raise-continuable obj)`

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

`(rational? obj)`

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

`(rationalize x y)`

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

`(read-bytevector k [port])`

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

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

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

`(read-char [port])`

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

`(read-error? obj)`

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

`(read-line [port])`

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

`(read-string k [port])`

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

`(read-u8 [port])`

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

`(real? obj)`

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

`(remainder n1 n2)`

TODO

`(reverse list)`

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

`(round x)`

TODO

`(set! <variable> <expression>)` syntax

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

`(set-car! pair obj)`

Stores `obj` in the car field of `pair`.

`(set-cdr! pair obj)`

Stores obj in the cdr field of pair.

`(square z)`

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

`(string char ...)`

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

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

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

`(string->number string [radix])`

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

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

`(string->symbol string)`

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

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

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

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

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

`(string-append string ...)`

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

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

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

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

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

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

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

It is an error if fill is not a character.

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

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

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

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

`(string-length string)`

Returns the number of characters in the given string.

`(string-map proc string1 ...)`

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

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

(string-ref string k)

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

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

(string-set! string k char)

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

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

string<=?

TODO

string<?

TODO

(string=? string1 string2 ...)

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

string>=?

TODO

string>?

TODO

(string? obj)

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

(substring string start end)

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

`(symbol->string symbol)`

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

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

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

`(symbol? obj)`

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

`syntax-error`

TODO

`syntax-rules`

TODO

`textual-port?`

TODO

`(truncate x)`

TODO

`truncate-quotient`

TODO

`truncate-remainder`

TODO

`truncate/`

TODO

`(u8-ready? [port])`

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

`(unless <test> <expr> ...)` syntax

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

`unquote`

TODO

`unquote-splicing`

TODO

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

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

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

`(values obj ...)`

Delivers all of its arguments to its continuation.

`(vector obj ...)`

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

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

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

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

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

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

`(vector-append vector ...)`

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

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

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

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

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

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

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

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

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

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

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

`(vector-length vector)`

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

`(vector-map proc vector1 ...)`

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

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

elements of the vectors is unspecified. If multiple returns occur from vector-map, the values returned by earlier returns are not mutated.

`(vector-ref vector k)`

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

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

`(vector-set! vector k obj)`

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

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

`vector?`

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

`(when <test> <expr> ...)` syntax

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

`with-exception-handler`

TODO

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

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

`(write-char char [port])`

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

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

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

`(write-u8 byte [port])`

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

(zero? z)

Return `#t` if z is zero. Otherwise `#f`. # `(scheme 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 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 sort)`

This is based on SRFI-132.

This library describes the API for a full-featured sort toolkit.

(list-sorted? < lis)

(vector-sorted? < v [start [end]]

These procedures return true iff their input list or vector is in sorted order, as determined by <. Specifically, they return #f iff there is an adjacent pair … X Y … in the input list or vector such that Y < X in the sense of <. The optional start and end range arguments restrict vector-sorted? to examining the indicated subvector.

These procedures are equivalent to the SRFI 95 sorted? procedure when applied to lists or vectors respectively, except that they do not accept a key procedure.

(list-sort < lis)

(list-stable-sort < lis)

These procedures do not alter their inputs, but are allowed to return a value that shares a common tail with a list argument.

The list-stable-sort procedure is equivalent to the R6RS list-sort procedure. It is also equivalent to the SRFI 95 sort procedure when applied to lists, except that it does not accept a key procedure.

(list-sort! < lis)

(list-stable-sort! < lis)

These procedures are linear update operators — they are allowed, but not required, to alter the cons cells of their arguments to produce their results. They return a sorted list containing the same elements as lis.

The list-stable-sort! procedure is equivalent to the SRFI 95 sort! procedure when applied to lists, except that it does not accept a key procedure.

(vector-sort < v [start [end]])

(vector-stable-sort < v [start [end]])

These procedures do not alter their inputs, but allocate a fresh vector as their result, of length end - start. The vector-stable-sort procedure with no optional arguments is equivalent to the R6RS vector-sort procedure. It is also equivalent to the SRFI 95 sort procedure when applied to vectors, except that it does not accept a key procedure.

(vector-sort! < v [start [end]])

(vector-stable-sort! < v [start [end]])

These procedures sort their data in-place. (But note that vector-stable-sort! may allocate temporary storage proportional to the size of the input — there are no known O(n lg n) stable vector sorting algorithms that run in constant space.) They return an unspecified value.

The vector-sort! procedure with no optional arguments is equivalent to the R6RS vector-sort! procedure.

(list-merge < lis1 lis2)

This procedure does not alter its inputs, and is allowed to return a value that shares a common tail with a list argument.

This procedure is equivalent to the SRFI 95 merge procedure when applied to lists, except that it does not accept a key procedure.

(list-merge! < lis1 lis2)

This procedure makes only a single, iterative, linear-time pass over its argument lists, using set-cdr!s to rearrange the cells of the lists into the list that is returned — it works "in place." Hence, any cons cell appearing in the result must have originally appeared in an input. It returns the sorted input.

Additionally, list-merge! is iterative, not recursive — it can operate on arguments of arbitrary size without requiring an unbounded amount of stack space.

The intent of this iterative-algorithm commitment is to allow the programmer to be sure that if, for example, list-merge! is asked to merge two ten-million-element lists, the operation will complete without performing some extremely (possibly twenty-million) deep recursion.

This procedure is equivalent to the SRFI 95 merge! procedure when applied to lists, except that it does not accept a key procedure.

`(vector-merge < v1 v2 [ start1 [ end1 [ start2 [ end2 ] ] ] ])`

This procedure does not alter its inputs, and returns a newly allocated vector of length (end1 - start1) + (end2 - start2).

This procedure is equivalent to the SRFI 95 merge procedure when applied to vectors, except that it does not accept a key procedure.

`(vector-merge! < to from1 from2 [ start [ start1 [ end1 [ start2 [ end2 ] ] ] ] ])`

This procedure writes its result into vector to, beginning at index start, for indices less than end, which is defined as start + (end1 - start1) + (end2 - start2). The target subvector to[start, end) may not overlap either of the source subvectors from1[start1, end1] and from2[start2, end2]. It returns an unspecified value.

This procedure is equivalent to the SRFI 95 merge! procedure when applied to lists, except that it does not accept a key procedure.

`(list-delete-neighbor-dups = lis)`

This procedure does not alter its input list, but its result may share storage with the input list.

`(list-delete-neighbor-dups! = lis)`

This procedure mutates its input list in order to construct its result. It makes only a single, iterative, linear-time pass over its argument, using set-cdr!s to rearrange the cells of the list into the final result — it works "in place." Hence, any cons cell appearing in the result must have originally appeared in the input.

`(vector-delete-neighbor-dups = v [ start [ end ] ])`

This procedure does not alter its input vector, but rather newly allocates and returns a vector to hold the result.

(vector-delete-neighbor-dups! = v [start [end]])

This procedure reuses its input vector to hold the answer, packing it into the index range [start, newend), where newend is the non-negative exact integer that is returned as its value. The vector is not altered outside the range [start, newend).

Examples:

```
(list-delete-neighbor-dups = '(1 1 2 7 7 7 0 -2 -2))
          => (1 2 7 0 -2)

(vector-delete-neighbor-dups = '#(1 1 2 7 7 7 0 -2 -2))
          => #(1 2 7 0 -2)

(vector-delete-neighbor-dups < '#(1 1 2 7 7 7 0 -2 -2) 3 7))
          => #(7 0 -2)

;; Result left in v[3,9):
(let ((v (vector 0 0 0 1 1 2 2 3 3 4 4 5 5 6 6)))
  (cons (vector-delete-neighbor-dups! < v 3)
        v))
              => (9 . #(0 0 0 1 2 3 4 5 6 4 4 5 5 6 6))
```

(vector-find-median < v knil [mean])

This procedure does not alter its input vector, but rather newly allocates a vector to hold the intermediate result. Runs in O(n) time.

(vector-find-median! < v knil [mean])

This procedure reuses its input vector to hold the intermediate result, leaving it sorted, but is otherwise the same as vector-find-median. Runs in O(n ln n) time.

(vector-select! < v k [start [end]])

This procedure returns the kth smallest element (in the sense of the < argument) of the region of a vector between start and end. Elements within the range may be reordered, whereas those outside the range are left alone. Runs in O(n) time.

(vector-separate! < v k [start [end]])

This procedure places the smallest k elements (in the sense of the < argument) of the region of a vector between start and end into the first k positions of that range, and the remaining elements into the remaining positions. Otherwise, the elements are not in any particular order. Elements outside the range are left alone. Runs in O(n) time. Returns an unspecified value. # `(scheme read)`

(read [port])

The `read` procedure converts external epresentations 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 ilist)`

This library is based on SRFI-116.

Scheme currently does not provide immutable pairs corresponding to its existing mutable pairs, although most uses of pairs do not exploit their mutability. The Racket system takes the radical approach of making Scheme's pairs immutable, and providing a minimal library of mutable pairs with procedures named mpair?, mcons, mcar, mcdr, set-mcar!, set-mcdr!. This SRFI takes the opposite approach of leaving Scheme's pairs unchanged and providing a full set of routines for creating and dealing with immutable pairs. The sample implementation is portable (to systems with SRFI 9) and efficient.

(ipair a d)

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

```
(ipair 'a '())        => (a)
(ipair (iq a) (iq b c d)) => ((a) b c d)
(ipair "a" (iq b c))    => ("a" b c)
(ipair 'a 3)          => (a . 3)
(ipair (iq a b) 'c)     => ((a b ) . c)
```

(ilist object ...)

Returns a newly allocated ilist of its arguments.

```
(ilist 'a (+ 3 4) 'c) =>  (a 7 c)
(ilist)               =>  ()
```

(xipair d a)

```
(lambda (d a) (ipair a d))
```

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

```
(xipair (iq b c) 'a) => (a b c)
```

The name stands for "eXchanged Immutable PAIR."

‘(ipair* elt1 elt2 ...)

Like ilist, but the last argument provides the tail of the constructed ilist, returning

```
(ipair elt1 (ipair elt2 (ipair ... eltn)))
```

```
(ipair* 1 2 3 4) => (1 2 3 . 4)
(ipair* 1) => 1
```

(make-ilist n [fill])

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

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

(ilist-tabulate n init-proc)

Returns an n-element ilist. Element i of the ilist, 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.

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

(ilist-copy dilist)

Copies the spine of the argument, including the ilist tail.

(iiota count [start step])

Returns an ilist containing the elements

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

The start and step parameters default to 0 and 1, respectively. This procedure takes its name from the APL primitive.

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

(proper-ilist? x)

(ilist? x)

These identifiers are bound either to the same procedure, or to procedures of equivalent behavior. In either case, true is returned iff x is a proper ilist — a ()-terminated ilist.

More carefully: The empty list is a proper ilist. An ipair whose icdr is a proper ilist is also a proper ilist. Everything else is a dotted ilist. This includes non-ipair, non-() values (e.g. symbols, numbers, mutable pairs), which are considered to be dotted ilists of length 0.

(dotted-ilist? x)

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

```
(dotted-ilist? x) = (not (proper-ilist? x))
```

(ipair? object)

Returns #t if object is an ipair; otherwise, #f.

```
(ipair? (ipair 'a 'b)) =>  #t
(ipair? (iq a b c)) =>  #t
(ipair? (cons 1 2)) =>  #f
(ipair? '())        =>  #f
(ipair? '#(a b))    =>  #f
(ipair? 7)          =>  #f
(ipair? 'a)         =>  #f
```

'(null-ilist? ilist)

Ilist is a proper ilist. 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 ilist. This procedure is recommended as the termination condition for ilist-processing procedures that are not defined on dotted ilists.

(not-ipair? x)

```
(lambda (x) (not (ipair? x)))
```

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

(ilist= elt= ilist1 ...)

Determines ilist equality, given an element-equality procedure. Proper ilist A equals proper ilist B if they are of the same length, and their corresponding elements are equal, as determined by elt=. If the element-comparison procedure's first argument is from ilisti, then its second argument is from ilisti+1, i.e. it is always called as (elt= a b) for a an element of ilist A, and b an element of ilist B.

In the n-ary case, every ilisti is compared to ilisti+1 (as opposed, for example, to comparing ilist1 to ilisti, for i>1). If there are no ilist arguments at all, ilist= simply returns true.

It is an error to apply ilist= to anything except proper ilists. It cannot reasonably be extended to dotted ilists, as it provides no way to specify an equality procedure for comparing the ilist terminators.

Note that the dynamic order in which the elt= procedure is applied to pairs of elements is not specified. For example, if ilist= is applied to three ilists, A, B, and C, it may first completely compare A to B, then compare B to C, or it may compare the first elements of A and B, then the first elements of B and C, then the second elements of A and B, and so forth.

The equality procedure must be consistent with eq?. That is, it must be the case that:

```
(eq? x y) => (elt= x y).
```

Note that this implies that two ilists which are eq? are always ilist=, as well; implementations may exploit this fact to "short-cut" the element-by-element comparisons.

```
(ilist= eq?) => #t       ; Trivial cases
(ilist= eq? (iq a)) => #t
```

(icar ipair)

(icdr ipair)

These procedures return the contents of the icar and icdr field of their argument, respectively. Note that it is an error to apply them to the empty ilist.

```
(icar (iq a b c))       =>  a        (icdr (iq a b c))     =>  (b c)
(icar (iq (a) b c d))   =>  (a)      (icdr (iq (a) b c d)) =>  (b c d)
(icar (ipair 1 2))      =>  1        (icdr (ipair 1 2))    =>  2
(icar '())              =>  *error*  (icdr '())            =>  *error*
```

(icaar ipair)

(icadr ipair)

…

(icdddar ipair)

'(icddddr ipair)

These procedures are compositions of icar and icdr, where for example icaddr could be defined by

```
(define icaddr (lambda (x) (icar (icdr (icdr x)))))
```

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

(ilist-ref ilist i)

Returns the ith element of ilist. (This is the same as the icar of (idrop ilist i).) It is an error if i >= n, where n is the length of ilist.

```
(ilist-ref (iq a b c d) 2) => c
```

(ifirst ipair)

(isecond ipair)

(ithird ipair)

(ifourth ipair)

(ififth ipair)

(isixth ipair)

(iseventh ipair)

(ieighth ipair)

(ininth ipair)

(itenth ipair)

Synonyms for car, cadr, caddr, ...

```
(ithird '(a b c d e)) => c
```

(icar+icdr ipair)

The fundamental ipair deconstructor:

```
(lambda (p) (values (icar p) (icdr p)))
```

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

(itake x i)

(idrop x i)

(ilist-tail x i)

`itake` returns the first i elements of ilist x.

`idrop` returns all but the first i elements of ilist x.

`ilist-tail` is either the same procedure as idrop or else a procedure with the same behavior.

“scheme (itake (iq a b c d e) 2) => (a b) (idrop (iq a b c d e) 2) => (c d e)

`x may be any value - a proper or dotted ilist:`

```scheme
(itake (ipair 1 (ipair 2 (ipair 3 'd)))    => (1 2)
(idrop (ipair 1 (ipair 2 (ipair 3 'd))) 2) => (3 . d)
(itake (ipair 1 (ipair 2 (ipair 3 'd))) 3) => (1 2 3)
(idrop (ipair 1 (ipair 2 (ipair 3 'd))) 3) => d
```

For a legal i, itake and idrop partition the ilist in a manner which can be inverted with iappend:

`(iappend (itake x i) (idrop x i)) = x`

idrop is exactly equivalent to performing i icdr operations on x; the returned value shares a common tail with x.

(itake-right dilist i)

(idrop-right dilist i)

itake-right returns the last i elements of dilist. idrop-right returns all but the last i elements of dilist.

```
(itake-right (iq a b c d e) 2) => (d e)
(idrop-right (iq a b c d e) 2) => (a b c)
```

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

dilist may be any ilist, either proper or dotted:

```
(itake-right (iq ipair 1 (ipair 2 (ipair 3 'd))) 2) => (2 3 . d)
(idrop-right (ipair 1 (ipair 2 (ipair 3 'd))) 2)    => (1)
(itake-right (ipair 1 (ipair 2 (ipair 3 'd))) 0)    => d
(idrop-right (ipair 1 (ipair 2 (ipair 3 'd))) 0)    => (1 2 3)
```

For a legal i, itake-right and idrop-right partition the ilist in a manner which can be inverted with iappend:

`(iappend (itake dilist i) (idrop dilist i)) = dilist`

itake-right's return value is guaranteed to share a common tail with dilist.

(isplit-at x i)

isplit-at splits the ilist x at index i, returning an ilist of the first i elements, and the remaining tail. It is equivalent to

```
(values (itake x i) (idrop x i))
```

(ilast ipair)

(last-ipair ipair)

Returns the last element of the non-empty, possibly dotted, ilist ipair. last-ipair returns the last ipair in the non-empty ilist pair.

```
(ilast (iq a b c))      => c
(last-ipair (iq a b c)) => (c)
```

(ilength ilist)

Returns the length of its argument. It is an error to pass a value to ilength which is not a proper ilist (()-terminated).

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

(iappend ilist1 ...)

Returns an ilist consisting of the elements of ilist1 followed by the elements of the other ilist parameters.

```
(iappend (iq x) (iq y))        =>  (x y)
(iappend (iq a) (iq b c d))    =>  (a b c d)
(iappend (iq a (b)) (iq (c)))  =>  (a (b) (c))
```

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

```
(iappend (iq a b) (ipair 'c 'd))  =>  (a b c . d)
(iappend '() 'a)           =>  a
(iappend (iq x y))         =>  (x y)
(iappend)                  =>  ()
```

(iconcatenate ilist-of-ilists)

Appends the elements of its argument together. That is, iconcatenate returns

```
(iapply iappend ilist-of-ilists)
```

or, equivalently,

```
(ireduce-right iappend '() ilist-of-ilists)
```

Note that some Scheme implementations do not support passing more than a certain number (e.g., 64) of arguments to an n-ary procedure. In these implementations, the (iapply iappend …) idiom would fail when applied to long lists, but iconcatenate would continue to function properly.

As with iappend, the last element of the input list may be any value at all.

(ireverse ilist)

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

```
(ireverse (iq a b c)) =>  (c b a)
(ireverse (iq a (b c) d (e (f))))
        =>  ((e (f)) d (b c) a)
```

(iappend-reverse rev-head tail)

iappend-reverse returns (iappend (ireverse 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 ilist, 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 iappend-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.)

(izip ilist1 ilist2 ...)

```
(lambda ilists (iapply imap ilist ilists))
```

If izip is passed n ilists, it returns an ilist as long as the shortest of these ilists, each element of which is an n-element ilist comprised of the corresponding elements from the parameter ilists.

```
(izip (iq one two three)
  (iq 1 2 3)
  (iq odd even odd even odd even odd even))
   ;; => ((one 1 odd) (two 2 even) (three 3 odd))

(izip (iq 1 2 3)) => ((1) (2) (3))
```

(iunzip1 ilist)

(iunzip2 ilist)

(iunzip3 ilist)

(iunzip4 ilist)

(iunzip5 ilist)

iunzip1 takes an ilist of ilists, where every ilist must contain at least one element, and returns an ilist containing the initial element of each such ilist. That is, it returns (imap icar ilists). iunzip2 takes an ilist of ilists, where every ilist must contain at least two elements, and returns two values: an ilist of the first elements, and an ilist of the second elements. iunzip3 does the same for the first three elements of the ilists, and so forth.

```
(iunzip2 (iq (1 one) (2 two) (3 three))) =>
  (1 2 3)
  (one two three)
```

(icount pred ilist1 ilist2 ...)

pred is a procedure taking as many arguments as there are ilists and returning a single value. It is applied element-wise to the elements of the ilists, 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 ilist elements in a left-to-right order. The counting stops when the shortest ilist expires.

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

(ifold kons knil ilist1 ilist2 ...)

The fundamental ilist iterator.

First, consider the single ilist-parameter case. If ilist1 = (e1 e2 … en), then this procedure returns

```
(kons en ... (kons e2 (kons e1 knil)) ... )
```

That is, it obeys the (tail) recursion

```
(ifold kons knil lis) = (ifold kons (kons (icar lis) knil) (icdr lis))
(ifold kons knil '()) = knil
```

Examples:

```
(ifold + 0 lis)          ; Add up the elements of LIS.

(ifold ipair '() lis)        ; Reverse LIS.

(ifold ipair tail rev-head) ; See APPEND-REVERSE.

;; How many symbols in LIS?
(ifold (lambda (x count) (if (symbol? x) (+ count 1) count))
```

```
       0
       lis)

;; Length of the longest string in LIS:
(ifold (lambda (s max-len) (max max-len (string-length s)))
       0
       lis)
```

If n ilist arguments are provided, then the kons function must take n+1 parameters: one element from each ilist, and the "seed" or fold state, which is initially knil. The fold operation terminates when the shortest ilist runs out of values:

```
(ifold ipair* '() (iq a b c) (iq 1 2 3 4 5)) => (c 3 b 2 a 1)
```

(ifold-right kons knil ilist1 ilist2 ...)

The fundamental ilist recursion operator.

First, consider the single ilist-parameter case. If ilist1 = (e1 e2 … en), then this procedure returns

```
(kons e1 (kons e2 ... (kons en knil)))
```

That is, it obeys the recursion

```
(ifold-right kons knil lis) = (kons (icar lis) (ifold-right kons knil (icdr lis)))
(ifold-right kons knil '()) = knil
```

Examples:

```
(ifold-right ipair '() lis)     ; Copy LIS.

;; Filter the even numbers out of LIS.
(ifold-right (lambda (x l) (if (even? x) (ipair x l) l)) '() lis))
```

If n ilist arguments are provided, then the kons procedure must take n+1 parameters: one element from each ilist, and the "seed" or fold state, which is initially knil. The fold operation terminates when the shortest ilist runs out of values:

```
(ifold-right ipair* '() (iq a b c) (iq 1 2 3 4 5)) => (a 1 b 2 c 3)
```

(ipair-fold kons knil ilist1 ilist2 ...)

Analogous to fold, but kons is applied to successive sub-ilists of the ilists, rather than successive elements — that is, kons is applied to the ipairs making up the lists, giving this (tail) recursion:

```
(ipair-fold kons knil lis) = (let ((tail (icdr lis)))
                               (ipair-fold kons (kons lis knil) tail))
(ipair-fold kons knil '()) = knil
```

Example:

```
(ipair-fold ipair '() (iq a b c)) => ((c) (b c) (a b c))
```

(ipair-fold-right kons knil ilist1 ilist2 ...)

Holds the same relationship with ifold-right that ipair-fold holds with ifold. Obeys the recursion

```
(ipair-fold-right kons knil lis) =
    (kons lis (ipair-fold-right kons knil (icdr lis)))
(ipair-fold-right kons knil '()) = knil
```

Example:

```
(ipair-fold-right ipair '() (iq a b c)) => ((a b c) (b c) (c))
```

(ireduce f ridentity ilist)

ireduce is a variant of ifold.

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

```
(f x ridentity) = x
```

ireduce has the following definition:

If ilist = (), return ridentity;

Otherwise, return (ifold f (icar ilist) (icdr ilist)).

...in other words, we compute (ifold f ridentity ilist).

Note that ridentity is used only in the empty-list case. You typically use ireduce when applying f is expensive and you'd like to avoid the extra application incurred when ifold applies f to the head of ilist 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, ifold is useful in many contexts where ireduce is not (consider the examples given in the ifold definition — only one of the five folds uses a function with a right identity. The other four may not be performed with ireduce).

```
;; take the max of an ilist of non-negative integers.
(ireduce max 0 nums) ; i.e., (iapply max 0 nums)
```

(ireduce-right f ridentity ilist)

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

```
(ireduce-right f ridentity '()) = ridentity
(ireduce-right f ridentity (iq e1)) = (f e1 ridentity) = e1
```

```
(ireduce-right f ridentity (iq e1 e2 ...)) =
  (f e1 (ireduce f ridentity (e2 ...)))
```

...in other words, we compute (ifold-right f ridentity ilist).

```
;; Append a bunch of ilists together.
;; I.e., (iapply iappend ilist-of-ilists)
(ireduce-right iappend '() ilist-of-ilists)
```

(iunfold p f g seed [tail-gen])

iunfold is best described by its basic recursion:

```
(iunfold p f g seed) =
    (if (p seed) (tail-gen seed)
        (ipair (f seed)
               (iunfold p f g (g seed))))
```

- p, Determines when to stop unfolding.
- f, Maps each seed value to the corresponding ilist element.
- g, Maps each seed value to next seed value.
- seed, The "state" value for the unfold.
- tail-gen, Creates the tail of the ilist; defaults to (lambda (x) '())

In other words, we use g to generate a sequence of seed values

```
seed, g(seed), g2(seed), g3(seed), ...
```

These seed values are mapped to ilist elements by f, producing the elements of the result ilist in a left-to-right order. P says when to stop.

iunfold is the fundamental recursive ilist constructor, just as ifold-right is the fundamental recursive ilist consumer. While iunfold may seem a bit abstract to novice functional programmers, it can be used in a number of ways:

```
;; Ilist of squares: 1^2 ... 10^2
(iunfold (lambda (x) (> x 10))
         (lambda (x) (* x x))
    (lambda (x) (+ x 1))
    1)

(iunfold null-ilist? icar icdr lis) ; Copy a proper ilist.

;; Read current input port into an ilist of values.
(iunfold eof-object? values (lambda (x) (read)) (read))

;; Copy a possibly non-proper ilist:
(iunfold not-ipair? icar icdr lis
             values)
```

```
;; Append HEAD onto TAIL:
(iunfold null-ilist? icar icdr head
              (lambda (x) tail))
```

Interested functional programmers may enjoy noting that ifold-right and iunfold are in some sense inverses. That is, given operations knull?, kar, kdr, kons, and knil satisfying

```
(kons (kar x) (kdr x)) = x and (knull? knil) = #t
```

then

```
(ifold-right kons knil (iunfold knull? kar kdr x)) = x
```

and

```
(iunfold knull? kar kdr (ifold-right kons knil x)) = x
```

This combinator sometimes is called an "anamorphism;" when an explicit tail-gen procedure is supplied, it is called an "apomorphism."

(iunfold-right p f g seed [tail])

iunfold-right constructs an ilist with the following loop:

```
(let lp ((seed seed) (lis tail))
  (if (p seed) lis
      (lp (g seed)
          (ipair (f seed) lis))))


p
    Determines when to stop unfolding.
f
    Maps each seed value to the corresponding ilist element.
g
    Maps each seed value to next seed value.
seed
    The "state" value for the unfold.
tail
    ilist terminator; defaults to '().
```

In other words, we use g to generate a sequence of seed values

```
seed, g(seed), g2(seed), g3(seed), ...
```

These seed values are mapped to ilist elements by f, producing the elements of the result ilist in a right-to-left order. P says when to stop.

iunfold-right is the fundamental iterative ilist constructor, just as ifold is the fundamental iterative ilist consumer. While iunfold-right may seem a bit abstract to novice functional programmers, it can be used in a number of ways:

```
;; Ilist of squares: 1^2 ... 10^2
(iunfold-right zero?
               (lambda (x) (* x x))
               (lambda (x) (- x 1))
               10)

;; Reverse a proper ilist.
(iunfold-right null-ilist? icar icdr lis)

;; Read current input port into an ilist of values.
(iunfold-right eof-object? values (lambda (x) (read)) (read))

;; (iappend-reverse rev-head tail)
(iunfold-right null-ilist? icar icdr rev-head tail)
```

Interested functional programmers may enjoy noting that ifold and iunfold-right are in some sense inverses. That is, given operations knull?, kar, kdr, kons, and knil satisfying

`(kons (kar x) (kdr x)) = x and (knull? knil) = #t`

then

`(ifold kons knil (iunfold-right knull? kar kdr x)) = x`

and

`(iunfold-right knull? kar kdr (ifold kons knil x)) = x.`

This combinator presumably has some pretentious mathematical name; interested readers are invited to communicate it to the author.

(imap proc ilist1 ilist2 ...)

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

```
(imap icadr (iq (a b) (d e) (g h))) =>  (b e h)

(imap (lambda (n) (expt n n))
      (iq 1 2 3 4 5))
    =>  (1 4 27 256 3125)

(imap + (iq 1 2 3) (iq 4 5 6)) =>  (5 7 9)

(let ((count 0))
  (imap (lambda (ignored)
          (set! count (+ count 1))
```

```
        count)
      (iq a b))) =>  (1 2) or (2 1)
```

(ifor-each proc ilist1 ilist2 ...)

The arguments to ifor-each are like the arguments to imap, but ifor-each calls proc for its side effects rather than for its values. Unlike imap, ifor-each is guaranteed to call proc on the elements of the ilists in order from the first element(s) to the last, and the value returned by ifor-each is unspecified.

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

(iappend-map f ilist1 ilist2 ...)

Equivalent to

```
(iapply iappend (imap f ilist1 ilist2 ...))
```

and

```
(iapply iappend (imap f ilist1 ilist2 ...))
```

Map f over the elements of the ilists, just as in the imap function. However, the results of the applications are appended together (using iappend) to make the final result.

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

Example:

```
(iappend-map (lambda (x) (ilist x (- x))) (iq 1 3 8))
  ;; => (1 -1 3 -3 8 -8)
```

(imap-in-order f ilist1 ilist2 ...)

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

(ipair-for-each f ilist1 ilist2 ...)

Like ifor-each, but f is applied to successive sub-ilists of the argument ilists. That is, f is applied to the cells of the ilists, rather than the ilists' elements. These applications occur in left-to-right order.

```
(ipair-for-each (lambda (ipair) (display ipair) (newline)) (iq a b c)) ==>
    (a b c)
    (b c)
    (c)
```

(ifilter-map f ilist1 ilist2 ...)

Like imap, but only true values are saved.

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

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

(ifilter pred ilist)

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

```
(ifilter even? (iq 0 7 8 8 43 -4)) => (0 8 8 -4)
```

(ipartition pred ilist)

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

```
(ipartition symbol? (iq one 2 3 four five 6)) =>
    (one four five)
    (2 3 6)
```

(iremove pred ilist)

Returns ilist without the elements that satisfy predicate pred:

```
(lambda (pred ilist) (ifilter (lambda (x) (not (pred x))) ilist))
```

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

```
(iremove even? (iq 0 7 8 8 43 -4)) => (7 43)
```

(ifind pred ilist)

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

```
(ifind even? (iq 3 1 4 1 5 9)) => 4
```

Note that ifind has an ambiguity in its lookup semantics — if ifind 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 ilist being searched is known not to contain any #f elements, or the ilist is guaranteed to have an element satisfying pred. However, in cases where this ambiguity can arise, you should use ifind-tail instead of ifind — ifind-tail has no such ambiguity:

```
(cond ((ifind-tail pred lis) => (lambda (ipair) ...)) ; Handle (icar ipair)
      (else ...)) ; Search failed.
```

(ifind-tail pred ilist)

Return the first ipair of ilist whose icar satisfies pred. If no ipair does, return false.

ifind-tail can be viewed as a general-predicate variant of the imember function.

Examples:

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

;; IMEMBER X LIS:
(ifind-tail (lambda (elt) (equal? x elt)) lis)
```

iqfind-tail is essentially idrop-while, where the sense of the predicate is inverted: Ifind-tail searches until it finds an element satisfying the predicate; idrop-while searches until it finds an element that doesn't satisfy the predicate.

(itake-while pred ilist)

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

```
(itake-while even? (iq 2 18 3 10 22 9)) => (2 18)
```

(idrop-while pred ilist)

idrops the longest initial prefix of ilist whose elements all satisfy the predicate pred, and returns the rest of the ilist.

```
(idrop-while even? (iq 2 18 3 10 22 9)) => (3 10 22 9)
```

(ispan pred ilist)

(ibreak pred ilist)

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

In other words: ispan finds the initial span of elements satisfying pred, and ibreak breaks the ilist at the first element satisfying pred.

ispan is equivalent to

```
(values (itake-while pred ilist)
        (idrop-while pred ilist))

(ispan even? (iq 2 18 3 10 22 9)) =>
  (2 18)
  (3 10 22 9)

(ibreak even? (iq 3 1 4 1 5 9)) =>
  (3 1)
  (4 1 5 9)
```

(iany pred ilist1 ilist2 ...)

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

If there are n ilist arguments ilist1 … ilistn, then pred must be a procedure taking n arguments and returning a boolean result.

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

Note the difference between ifind and iany — ifind returns the element that satisfied the predicate; iany returns the true value that the predicate produced.

Like ievery, iany'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.

```
(iany integer? (iq a 3 b 2.7))   => #t
(iany integer? (iq a 3.1 b 2.7)) => #f
(iany < (iq 3 1 4 1 5)
        (iq 2 7 1 8 2)) => #t
```

(ievery pred ilist1 ilist2 ...)

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

If there are n ilist arguments ilist1 … ilistn, then pred must be a procedure taking n arguments and returning a boolean result.

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

If one of the ilisti has no elements, ievery simply returns #t.

Like iany, ievery'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.

(ilist-index pred ilist1 ilist2 ...)

Return the index of the leftmost element that satisfies pred.

If there are n ilist arguments ilist1 … ilistn, then pred must be a function taking n arguments and returning a boolean result.

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

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

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

(imember x ilist [=])

(imemq x ilist)

'(imemv x ilist)

These procedures return the first sub-ilist of ilist whose icar is x, where the sub-ilists of ilist are the non-empty ilists returned by (idrop ilist i) for i less than the length of ilist. If x does not occur in ilist, then #f is returned. imemq uses eq?

to compare x with the elements of ilist, while imemv uses eqv?, and imember uses equal?.

```
(imemq 'a (iq a b c))           =>  (a b c)
(imemq 'b (iq a b c))           =>  (b c)
(imemq 'a (iq b c d))           =>  #f
(imemq (list 'a)
       (ilist 'b '(a) 'c))      =>  #f
(imember (list 'a)
       (ilist 'b '(a) 'c)))     =>  ((a) c)
(imemq 101 (iq 100 101 102))    =>  *unspecified*
(imemv 101 (iq 100 101 102))    =>  (101 102)
```

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

```
(= x ei) ; ilist is (E1 ... En)
```

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

Note that fully general ilist searching may be performed with the ifind-tail and ifind procedures, e.g.

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

(idelete x ilist [=])

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

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

Note that fully general element deletion can be performed with the iremove procedures, e.g.:

```
;; idelete all the even elements from LIS:
(iremove even? lis)
```

The comparison procedure is used in this way: (= x ei). That is, x is always the first argument, and an ilist element is always the second argument. The comparison procedure will be used to compare each element of ilist 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 an ilist with (idelete 5 ilist <)

(idelete-duplicates ilist [=])

idelete-duplicates removes duplicate elements from the ilist argument. If there are multiple equal elements in the argument ilist, the result ilist 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 ilist — idelete-duplicates does not disorder the ilist (hence it is useful for "cleaning up" immutable association lists).

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

Implementations of idelete-duplicates are allowed to share common tails between argument and result ilists — for example, if the ilist argument contains only unique elements, it may simply return exactly this ilist.

Be aware that, in general, idelete-duplicates runs in time O(n2) for n-element ilists. Uniquifying long ilists can be accomplished in O(n lg n) time by sorting the ilist 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.

```
(idelete-duplicates (iq a b a c a b c z)) => (a b c z)

;; Clean up an ialist:
(idelete-duplicates (iq (a . 3) (b . 7) (a . 9) (c . 1))
  (lambda (x y) (eq? (icar x) (icar y))))
;; => ((a . 3) (b . 7) (c . 1))
```

(ialist-cons key datum ialist)

```
(lambda (key datum ialist) (ipair (ipair key datum) ialist))
```

Construct a new ialist entry mapping key -> datum onto ialist.

(ialist-delete key ialist [=])

ialist-delete deletes all associations from ialist 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 ialist argument. The ialist is not disordered — elements that appear in the result ialist occur in the same order as they occur in the argument ialist.

The comparison procedure is used to compare the element keys ki of ialist's entries to the key parameter in this way: (= key ki). Thus, one can reliably

remove all entries of ialist whose key is greater than five with (ialist-delete 5 ialist <)

(replace-icar ipair object)

This procedure returns an ipair with object in the icar field and the icdr of ipair in the icdr field.

(replace-icdr ipair object)

This procedure returns an ipair with object in the icdr field and the icar of ipair in the icar field.

(pair->ipair pair)

(ipair->pair ipair)

These procedures, which are inverses, return an ipair and a pair respectively that have the same (i)car and (i)cdr fields as the argument.

(list->ilist flist)

(ilist->list dilist)

These procedures return an ilist and a list respectively that have the same elements as the argument. The tails of dotted (i)lists are preserved in the result, which makes the procedures not inverses when the tail of a dotted ilist is a list or vice versa. The empty list is converted to itself.

It is an error to apply list->ilist to a circular list.

(tree->itree object)

(itree->tree object)

These procedures walk a tree of pairs or ipairs respectively and make a deep copy of it, returning an isomorphic tree containing ipairs or pairs respectively. The result may share structure with the argument. If the argument is not of the expected type, it is returned.

These procedures are not inverses in the general case. For example, a pair of ipairs would be converted by tree->itree to an ipair of ipairs, which if converted by itree->tree would produce a pair of pairs.

(gtree->itree object)

`(gtree->tree object)`

These procedures walk a generalized tree consisting of pairs, ipairs, or a combination of both, and make a deep copy of it, returning an isomorphic tree containing only ipairs or pairs respectively. The result may share structure with the argument. If the argument is neither a pair nor an ipair, it is returned.

`(iapply procedure object ... ilist)`

The iapply procedure is an analogue of apply whose last argument is an ilist rather than a list. It is equivalent to (apply procedure object … (ilist->list ilist)), but may be implemented more efficiently.

`ipair-comparator`

The ipair-comparator object is a SRFI-114 comparator suitable for comparing ipairs. Note that it is not a procedure. It compares pairs using default-comparator 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.

`ilist-comparator`

The ilist-comparator object is a SRFI-114 comparator suitable for comparing ilists. Note that it is not a procedure. It compares ilists lexicographically, as follows:

- The empty ilist compares equal to itself.
- The empty ilist compares less than any non-empty ilist.
- Two non-empty ilists are compared by comparing their icars. If the icars are not equal when compared using default-comparator, then the result is the result of that comparison. Otherwise, the icdrs are compared using ilist-comparator.

`(make-ilist-comparator comparator)`

The make-ilist-comparator procedure returns a comparator suitable for comparing ilists using element-comparator to compare the elements.

`(make-improper-ilist-comparator comparator)`

The make-improper-ilist-comparator procedure returns a comparator that compares arbitrary objects as follows: the empty list precedes all ipairs, which precede all other objects. Ipairs are compared as if with (make-ipair-comparator comparator comparator). All other objects are compared using comparator.

`(make-icar-comparator comparator)`

The make-icar-comparator procedure returns a comparator that compares ipairs on their icars alone using comparator.

‘(make-icdr-comparator comparator)

The make-icdr-comparator procedure returns a comparator that compares ipairs on their icdrs alone using comparator. # `(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 lseq)`

This library is based on SRFI-127.

Lazy sequences (or lseqs, pronounced “ell-seeks”) are a generalization of lists. In particular, an lseq is either a proper list or a dotted list whose last cdr is a SRFI 121 generator. A generator is a procedure that can be invoked with no arguments in order to lazily supply additional elements of the lseq. When

a generator has no more elements to return, it returns an end-of-file object. Consequently, lazy sequences cannot reliably contain end-of-file objects.

This SRFI provides a set of procedures suitable for operating on lazy sequences based on SRFI 1.

(generator->lseq generator)

Returns an lseq whose elements are the values generated by generator. The exact behavior is as follows:

- Generator is invoked with no arguments to produce an object obj.
- If obj is an end-of-file object, the empty list is returned.
- Otherwise, a newly allocated pair whose car is obj and whose cdr is generator is returned.

```
(generator->lseq (make-iota-generator +inf.0 1))
```

(lseq? x)

Returns #t if x is an lseq. This procedure may also return #t if x is an improper list whose last cdr is a procedure that requires arguments, since there is no portable way to examine a procedure to determine how many arguments it requires. Otherwise it returns #f.

(lseq=? elt=? lseq1 lseq2)

Determines lseq equality, given an element-equality procedure. Two lseqs are equal if they are of the same length, and their corresponding elements are equal, as determined by elt=?. When elt=? is called, its first argument is always from lseq1 and its second argument is from lseq2.

The dynamic order in which the elt=? procedure is applied to pairs of elements is not specified.

The elt=? procedure must be consistent with eq?. This implies that two lseqs which are eq? are always lseq=?, as well; implementations may exploit this fact to "short-cut" the element-by-element equality tests.

(lseq-car lseq)

(lseq-first lseq)

These procedures are synonymous. They return the first element of lseq. They are included for completeness, as they are the same as car. It is an error to apply them to an empty lseq.

(lseq-cdr lseq)

(lseq-rest lseq)

These procedures are synonymous. They return an lseq with the contents of lseq except for the first element. The exact behavior is as follows:

- If lseq is a pair whose cdr is a procedure, then the procedure is invoked with no arguments to produce an object obj.
- If obj is an end-of-file object, then the cdr of lseq is set to the empty list, which is returned.
- If obj is any other object, then a new pair is allocated whose car is obj and whose cdr is the cdr of lseq (i.e. the procedure). The cdr of lseq is set to the newly allocated pair, which is returned.
- If lseq is a pair whose cdr is not a procedure, then the cdr is returned.
- If lseq is not a pair, it is an error.

Implementations that inline cdr are advised to inline lseq-cdr if possible.

(lseq-ref lseq i)

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

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

(lseq-take lseq i)

(lseq-drop lseq i)

lseq-take lazily returns the first i elements of lseq. lseq-drop returns all but the first i elements of lseq.

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

lseq-drop is exactly equivalent to performing i lseq-rest operations on lseq.

(lseq-realize lseq)

Repeatedly applies lseq-cdr to lseq until its generator (if there is one) has been exhausted, and returns lseq, which is now guaranteed to be a proper list. This procedure can be called on an arbitrary lseq before passing it to a procedure which only accepts lists. However, if the generator never returns an end-of-file object, lseq-realize will never return.

(lseq->generator lseq)

Returns a generator which when invoked will return all the elements of lseq, including any that have not yet been realized.

(lseq-length lseq)

Returns the length of its argument, which is the non-negative integer n such that lseq-rest applied n times to the lseq produces an empty lseq. lseq must be finite, or this procedure will not return.

(lseq-append lseq ...)

Returns an lseq that lazily contains all the elements of all the lseqs in order.

(lseq-zip lseq1 lseq2 ...)

If lseq-zip is passed n lseqs, it lazily returns an lseq each element of which is an n-element list comprised of the corresponding elements from the lseqs. If any of the lseqs are finite in length, the result is as long as the shortest lseq.

```
(lseq-zip '(one two three)
     (generator->lseq (make-iota-generator +inf.0 1 1))
     (generator->lseq (make-repeating-generator) '(odd even))))
    => ((one 1 odd) (two 2 even) (three 3 odd))

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

(lseq-map proc lseq1 lseq2 ...)

The lseq-map procedure lazily applies proc element-wise to the corresponding elements of the lseqs, where proc is a procedure taking as many arguments as there are lseqs and returning a single value, and returns an lseq of the results in order. The dynamic order in which proc is applied to the elements of the lseqs is unspecified.

```
(lseq-map
  (lambda (x) (lseq-car (lseq-cdr x)))
  '((a b) (d e) (g h))) =>  (b e h)

(lseq-map (lambda (n) (expt n n))
     (make-iota-generator +inf.0 1 1)
    =>  (1 4 27 256 3125 ...)

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

(let ((count 0))
  (lseq-map (lambda (ignored)
```

```
    (set! count (+ count 1))
    count)
  '(a b))) =>  (1 2) or (2 1)
```

‘(lseq-for-each proc lseq1 lseq2 ...)

The arguments to lseq-for-each are like the arguments to lseq-map, but lseq-for-each calls proc for its side effects rather than for its values. Unlike lseq-map, lseq-for-each is guaranteed to call proc on the elements of the lseqs in order from the first element(s) to the last, and the value returned by lseq-for-each is unspecified.

If none of the lseqs are finite, lseq-for-each never returns.

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

(lseq-filter pred lseq)

(lseq-remove pred lseq)

The procedure lseq-filter lazily returns an lseq that contains only the elements of lseq that satisfy pred.

The procedure lseq-remove is the same as lseq-filter, except that it returns elements that do not satisfy pred. These procedures are guaranteed to call pred on the elements of the lseqs in sequence order.

```
(lseq-filter odd? (generator->lseq (make-range-generator 1 5)))
;; =>  (1 3)

(lseq-remove odd? (generator->lseq (make-range-generator 1 5)))
;; =>  (2 4)
```

(lseq-find-tail pred lseq)

Returns the longest tail of lseq whose first element satisfies pred, or #f if no element does. The predicate is guaranteed to be evaluated on the elements of lseq in sequence order, and only as often as necessary.

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

Examples:

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

;; equivalent to (lseq-member elt lseq)
(lseq-find-tail (lambda (elt) (equal? x elt)) lseq)
```

(lseq-take-while pred lseq)

Lazily returns the longest initial prefix of lseq whose elements all satisfy the predicate pred.

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

(lseq-drop-while pred lseq)

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

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

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

(lseq-any pred lseq1 lseq2 ...)

Applies pred to successive elements of the lseqs, returning true if pred returns true on any application. If an application returns a true value, lseq-any immediately returns that value. Otherwise, it iterates until a true value is produced or one of the lseqs runs out of values; in the latter case, lseq-any returns #f. It is an error if pred does not accept the same number of arguments as there are lseqs and return a boolean result.

Note the difference between lseq-find and lseq-any — lseq-find returns the element that satisfied the predicate; lseq-any returns the true value that the predicate produced.

Like lseq-every, lseq-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.

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

```
(define (factorial n)
  (cond
    ((< n 0) #f)
    ((= n 0) 1)
    (else (* n (factorial (- n 1))))))
(lseq-any factorial '(-1 -2 3 4))
        => 6
```

(lseq-every pred lseq1 lseq2 ...)

Applies pred to successive elements of the lseqs, returning true if the predicate returns true on every application. If an application returns a false value, lseq-every immediately returns that value. Otherwise, it iterates until a false value is produced or one of the lseqs runs out of values; in the latter case, lseq-every returns the last value returned by pred, or #t if pred was never invoked. It is an error if pred does not accept the same number of arguments as there are lseqs and return a boolean result.

Like lseq-any, lseq-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.

```
(lseq-every factorial '(1 2 3 4))
        => 24
```

(lseq-index pred lseq1 lseq2 ...)

Return the index of the leftmost element that satisfies pred.

Applies pred to successive elements of the lseqs, returning an index usable with lseq-ref if the predicate returns true. Otherwise, it iterates until one of the lseqs runs out of values, in which case #f is returned.

It is an error if pred does not accept the same number of arguments as there are lseqs and return a boolean result.

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

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

(lseq-member x lseq [pred])

(lseq-memq x lseq)

(lseq-memv x lseq)

These procedures return the longest tail of lseq whose first element is x, where the tails of lseq are the non-empty lseqs returned by (lseq-drop lseq i) for i less than the length of lseq. If x does not occur in lseq, then #f is returned. lseq-memq uses eq? to compare x with the elements of lseq, while lseq-memv uses eqv?, and lseq-member uses pred, which defaults to equal?.

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

The equality procedure is used to compare the elements ei of lseq to the key x in this way: the first argument is always x, and the second argument is one of the lseq elements. Thus one can reliably find the first element of lseq that is greater than five with (lseq-member 5 lseq <)

Note that fully general lseq searching may be performed with the lseq-find-tail procedure, e.g.

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

(scheme list-queue)

This library is based on SRFI-117.

List queues are mutable ordered collections that can contain any Scheme object. Each list queue is based on an ordinary Scheme list containing the elements of the list queue by maintaining pointers to the first and last pairs of the list. It's cheap to add or remove elements from the front of the list or to add elements to the back, but not to remove elements from the back. List queues are disjoint from other types of Scheme objects.

(make-list-queue list [last])

Returns a newly allocated list queue containing the elements of list in order. The result shares storage with list. If the last argument is not provided, this operation is O(n) where n is the length of list.

However, if last is provided, make-list-queue returns a newly allocated list queue containing the elements of the list whose first pair is first and whose last pair is last. It is an error if the pairs do not belong to the same list. Alternatively, both first and last can be the empty list. In either case, the operation is O(1).

Note: To apply a non-destructive list procedure to a list queue and return a new list queue, use (make-list-queue (proc (list-queue-list list-queue))).

`(list-queue element ...)`

Returns a newly allocated list queue containing the elements. This operation is O(n) where n is the number of elements.

`(list-queue-copy list-queue)`

Returns a newly allocated list queue containing the elements of list-queue. This operation is O(n) where n is the length of list-queue

`(list-queue-unfold stop? mapper successor seed [ queue ])`

Performs the following algorithm:

If the result of applying the predicate stop? to seed is true, return queue. Otherwise, apply the procedure mapper to seed, returning a value which is added to the front of queue. Then get a new seed by applying the procedure successor to seed, and repeat this algorithm.

If queue is omitted, a newly allocated list queue is used.

`(list-queue-unfold-right stop? mapper successor seed [ queue ])`

Performs the following algorithm:

If the result of applying the predicate stop? to seed is true, return the list queue. Otherwise, apply the procedure mapper to seed, returning a value which is added to the back of the list queue. Then get a new seed by applying the procedure successor to seed, and repeat this algorithm.

If queue is omitted, a newly allocated list queue is used.

`(list-queue? obj)`

Returns #t if obj is a list queue, and #f otherwise. This operation is O(1).

`(list-queue-empty? list-queue)`

Returns #t if list-queue has no elements, and #f otherwise. This operation is O(1).

(list-queue-front list-queue)

Returns the first element of list-queue. If the list queue is empty, it is an error. This operation is O(1).

(list-queue-back list-queue)

Returns the last element of list-queue. If the list queue is empty, it is an error. This operation is O(1).

(list-queue-list list-queue)

Returns the list that contains the members of list-queue in order. The result shares storage with list-queue. This operation is O(1).

(list-queue-first-last list-queue)

Returns two values, the first and last pairs of the list that contains the members of list-queue in order. If list-queue is empty, returns two empty lists. The results share storage with list-queue. This operation is O(1).

(list-queue-add-front! list-queue element)

Adds element to the beginning of list-queue. Returns an unspecified value. This operation is O(1).

(list-queue-add-back! list-queue element)

Adds element to the end of list-queue. Returns an unspecified value. This operation is O(1).

(list-queue-remove-front! list-queue)

Removes the first element of list-queue and returns it. If the list queue is empty, it is an error. This operation is O(1).

(list-queue-remove-back! list-queue)

Removes the last element of list-queue and returns it. If the list queue is empty, it is an error. This operation is O(n) where n is the length of list-queue, because queues do not not have backward links.

(list-queue-remove-all! list-queue)

Removes all the elements of list-queue and returns them in order as a list. This operation is O(1).

`(list-queue-set-list! list-queue list [ last ])`

Replaces the list associated with list-queue with list, effectively discarding all the elements of list-queue in favor of those in list. Returns an unspecified value. This operation is O(n) where n is the length of list. If last is provided, it is treated in the same way as in make-list-queue, and the operation is O(1).

Note: To apply a destructive list procedure to a list queue, use (list-queue-set-list! (proc (list-queue-list list-queue))).

`(list-queue-append list-queue ...)`

Returns a list queue which contains all the elements in front-to-back order from all the list-queues in front-to-back order. The result does not share storage with any of the arguments. This operation is O(n) in the total number of elements in all queues.

`(list-queue-append! list-queue ...)`

Returns a list queue which contains all the elements in front-to-back order from all the list-queues in front-to-back order. It is an error to assume anything about the contents of the list-queues after the procedure returns. This operation is O(n) in the total number of queues, not elements. It is not part of the R7RS-small list API, but is included here for efficiency when pure functional append is not required.

`(list-queue-concatenate list-of-list-queues)`

Returns a list queue which contains all the elements in front-to-back order from all the list queues which are members of list-of-list-queues in front-to-back order. The result does not share storage with any of the arguments. This operation is O(n) in the total number of elements in all queues. It is not part of the R7RS-small list API, but is included here to make appending a large number of queues possible in Schemes that limit the number of arguments to apply.

`(list-queue-map proc list-queue)`

Applies proc to each element of list-queue in unspecified order and returns a newly allocated list queue containing the results. This operation is O(n) where n is the length of list-queue.

`(list-queue-map! proc list-queue)`

Applies proc to each element of list-queue in front-to-back order and modifies list-queue to contain the results. This operation is O(n) in the length of list-queue. It is not part of the R7RS-small list API, but is included here to make transformation of a list queue by mutation more efficient.

(list-queue-for-each proc list-queue)

Applies proc to each element of list-queue in front-to-back order, discarding the returned values. Returns an unspecified value. This operation is O(n) where n is the length of list-queue. # `(scheme list)`

This is based on SRFI-1.

(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)
```

(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.

`(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)
```

(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 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.

(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.

(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.
```

(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))
```

(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.

(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.

(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*
```

(scheme hash-table)

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.

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 able.
- 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).

`(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.

`(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.

`(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

`(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.

`(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.

(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.

`(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-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. # `(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.

qIt 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 inexact)`

`(acos z)`

TODO

`(asin z)`

TODO

`(atan z)`

TODO

`(cos z)`

TODO

`(exp z t)`

TODO

`(finite? z)`

TODO

`(infinite? z)`

TODO

`(log z)`

TODO

`(nan? z)`

TODO

sin

TODO

sqrt

TODO

tan

TODO # `(scheme 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 show)

This library is based on SRFI-159.

A library of procedures for formatting Scheme objects to text in various ways, and for easily concatenating, composing and extending these formatters efficiently without resorting to capturing and manipulating intermediate strings.

(show output-dest fmt ...)

The entry point for all formatting. Applies the fmt formatters in sequence, accumulating the output to output-dest. As with SRFI 28 format, output-dest can be an output port, #t to indicate the current output port, or #f to accumulate the output into a string and return that as the result of show.

Each fmt should be a formatter as discussed below. As a convenience, non-formatter arguments are also allowed and are formatted as if wrapped with displayed, described below, so that

```
(show #f " π = " (with ((precision 2)) (acos -1)) nl)
```

would return the string "π = 3.14".

As mentioned, formatters are an opaque type and cannot directly be applied outside of show. Custom formatters are built on the existing formatters, and as first class objects may be named or computed dynamically, so that:

```
(let ((~.2f (lambda (x) (with ((precision 2)) x))))
  (show #f " π = " (~.2f (acos -1)) nl))
```

produces the same result. For typical uses you only need to combine the existing high level formatters described in the succeeding sections, but see the section Higher Order Formatters and State for control flow and state manipulation primitives.

The return value of show is the accumulated string if output-dest is #f and unspecified otherwise.

(displayed obj)

If obj is a formatter, returns obj as is. Otherwise, outputs obj using display semantics. Specifically, strings are output as if by write-string and characters are written as if by write-char. Other objects are output as with written (including nested strings and chars inside obj). This is the default behavior for top-level formats in show, each and most other high-level formatters.

(written obj)

Outputs obj using write semantics. Uses the current numeric formatting settings to the extent that the written result can still be passed to read, possibly with loss of precision. Specifically, the current radix is used if set to any of 2, 8, 10 or 16, and the fixed point precision is used if specified and the radix is 10.

```
(show #f (written (cons 0 1)))
=> "(0 . 1)"

(show #f 1.5 " " (with ((precision 0)) 1.5))
=> "1.5 2"

(show #f 1/7 " " (with ((precision 3)) 1/7)
      " " (with ((precision 20)) 1/7))
=> "1/7 0.143 0.14285714285714285714"
```

Implementations should allow arbitrary precision for exact rational numbers, for example, using string-segment from SRFI 152, the following code returns the first 100 Fibonacci numbers:

```
(map string->number
     (string-segment
      (show #f (with ((precision 2500))
                 (/ 1000 (- #e1e50 #e1e25 1))))
      25))
```

(written-simply obj)

As above, but doesn't handle shared structures. Infinite loops can still be avoided if used inside a formatter that truncates data (see trimmed and fitted below).

(pretty obj)

Pretty-prints obj. The result should be identical to written except possibly for differences in whitespace to make the output resemble formatted source code. Implementations should print vectors and data lists (lists that don't begin with a (nested) symbol) in a tabular format when possible to reduce vertical space.

(pretty-simply obj)

As above but without sharing.

(escaped str [quote-ch esc-ch renamer])

Outputs the string str, escaping any quote or escape characters. If esc-ch, which defaults to #\, is #f, escapes only the quote-ch, which defaults to #", by

doubling it, as in SQL strings and CSV values. If renamer is provided, it should be a procedure of one character which maps that character to its escape value, e.g. #
=> #, or #f if there is no escape value.

```
(show #f (escaped "hi, bob!"))
=> "hi, bob!"

(show #f (escaped "hi, \"bob!\""))
=> "hi, \"bob!\""
```

(maybe-escaped str pred [quote-ch esc-ch renamer])

Like escaped, but first checks if any quoting is required (by the existence of either any quote or escape characters, or any character matching pred), and if so outputs the string in quotes and with escapes. Otherwise outputs the string as is. This is useful for quoting symbols and CSV output, etc.

```
(show #f (maybe-escaped "foo" char-whitespace? #\"))
=> "foo"

(show #f (maybe-escaped "foo bar" char-whitespace? #\"))
=> "\"foo bar\""

(show #f (maybe-escaped "foo\"bar\"baz" char-whitespace? #\"))
=> "\"foo\"bar\"baz\""
```

(numeric num [radix precision sign comma comma-sep decimal-sep])

Formats a single number num. You can optionally specify any radix from 2 to 36 (even if num isn't an integer). precision forces a fixed-point format.

A sign of #t indicates to output a plus sign (+) for positive integers. However, if sign is a pair of two strings, it means to wrap negative numbers with the two strings. For example, ("(" . ")") prints negative numbers in parentheses, financial style: -1.99 => (1.99).

- comma is an integer specifying the number of digits between commas.
- comma-sep is the character to use for commas, defaulting to #,.
- decimal-sep is the character to use for decimals, defaulting to #., or to #, (European style) if comma-sep is already #..

These parameters may seem unwieldy, but they can also take their defaults from state variables, described below.

(numeric/comma num [base precision sign])

Shortcut for numeric to print with commas.

```
(show #f (numeric/comma 1234567))
=> "1,234,567"
```

(numeric/si num [base separator])

Abbreviates num with an SI suffix as in the -h or –si option to many GNU commands. The base defaults to 1024, using suffix names like Ki, Mi, Gi, etc. Other bases (e.g. the standard 1000) have the suffixes k, M, G, etc. If separator is provided, it is inserted after the number, before any suffix.

```
(show #f (numeric/si 608))
=> "608"

(show #f (numeric/si 608) "B")
=> "608B"

(show #f (numeric/si 608 1000 " ") "B")
=> "608 B"

(show #f (numeric/si 3986))
=> "3.9Ki"

(show #f (numeric/si 3986 1000) "B")
=> "4kB"

(show #f (numeric/si 1.23e-6 1000) "m")
=> "1.2µm"

(show #f (numeric/si 1.23e-6 1000 " ") "m")
=> "1.2 µm"

See https://en.wikipedia.org/wiki/Metric_prefix for the complete list of abbreviations.

    (numeric/fitted width n . args)

Like numeric, but if the result doesn't fit in width using the current precision, output ins

(show #f (with ((precision 2)) (numeric/fitted 4 1.25)))
=> "1.25"

(show #f (with ((precision 2)) (numeric/fitted 4 12.345)))
=> "#.##"
```

nl

Outputs a newline.

```
(show #f nl)
=> "\n"
```

fl

Short for "fresh line," outputs a newline only if we're not already at the start of a line.

```
(show #f fl)
=> ""

(show #f "hi" fl)
=> "hi\n"

(show #f "hi" nl fl)
=> "hi\n"
```

(space-to column)

Outputs spaces up to the given column. If the current column is already >= column, does nothing. The character used for spacing is the current value of pad-char, described below, which defaults to space. Columns are zero-based.

```
(show #f "a" (space-to 5) "b")
=> "a    b"

(show #f "a" (space-to 0) "b")
=> "ab"
```

(tab-to [tab-width])

Outputs spaces up to the next tab stop, using tab stops of width tab-width, which defaults to 8. If already on a tab stop, does nothing. If you want to ensure you always tab at least one space, you can use (each " " (tab-to width)). Columns are zero-based.

```
(show #f (tab-to 5) "b")
=> "b"

(show #f "a" (tab-to 5) "b")
=> "a    b"

(show #f "abcdefghi" (tab-to 5) "b")
=> "abcdefghi b"
```

nothing

Outputs nothing (useful in combinators and as a default noop in conditionals).

```
(show #f "a" nothing "b")
=> "ab"
```

(each fmt ...)

Applies each fmt in sequence, as in the top-level of show.

```
(show #f (each "a" "b"))
=> "ab"
```

(each-in-list list-of-fmts)

Equivalent to (apply each list-of-fmts) but may be more efficient.

(joined mapper list [sep])

Formats each element elt of list with (mapper elt), inserting sep in between. sep defaults to the empty string, but can be any format or string.

```
(show #f (joined displayed '(a b c) ", "))
=> "a, b, c"
```

(joined/prefix mapper list [sep])

(joined/suffix mapper list [sep])

```
(show #f (joined/prefix displayed '(usr local bin) "/"))
=> "/usr/local/bin"

(show #f (joined/suffix displayed '(1 2 3) nl))
=> "1\n2\n3\n"
```

As joined, but inserts sep before/after every element.

(joined/last mapper last-mapper list [sep])

As joined, but the last element of the list is formatted with last-mapper instead.

```
    (show #f (joined/last displayed
                          (lambda (last) (each "and " last))
                          '(lions tigers bears)
                          ", "))
    => "lions, tigers, and bears"
```

(joined/dot mapper dot-mapper list [sep])

As joined, but if the list is a dotted list, then formats the dotted value with dot-mapper instead.

```
(show #f
      "("
      (joined/dot displayed
          (lambda (dot) (each ". " dot))
          '(1 2 . 3)
          " ")
      ")")
=> "(1 2 . 3)"
```

(joined/range mapper start [end sep])

As joined, but counts from start (inclusive) to end (exclusive), formatting each integer in the range with mapper. If end is #f or unspecified, produces an infinite stream of output.

```
(show #f (joined/range displayed 0 5 " "))
=> "0 1 2 3 4"
```

(padded width fmt ...)

(padded/right width fmt ...)

(padded/both width fmt ...)

Analogs of SRFI 13 string-pad, these add extra space to the left, right or both sides of the output generated by the fmts to pad it to width. If width is exceeded, has no effect. padded/both will include one more extra space on the right side of the output if the difference is odd.

padded/right is guaranteed not to accumulate any intermediate data.

Note these are column-oriented padders, so won't necessarily work with multi-line output (padding doesn't seem a likely operation for multi-line output).

```
(show #f (padded 5 "abc"))
=> "  abc"

(show #f (padded/right 5 "abc"))
=> "abc  "

(show #f (padded/both 5 "abc"))
=> " abc "
```

(trimmed width fmt ...)

(trimmed/right width fmt ...)

(trimmed/both width fmt ...)

Analogs of SRFI 13 string-trim, these truncate the output of the fmts to force it in under width columns. As soon as any of the fmts exceeds width, stop formatting and truncate the result, returning control to whoever called trimmed. If width is not exceeded, is equivalent to each.

If a truncation ellipsis is set, then when any truncation occurs trimmed and trimmed/right will prepend and append the ellipsis, respectively. trimmed/both will both prepend and append. The length of the ellipsis will be considered when truncating the original string, so that the total width will never be longer than width. It is an error if width is less than the length of ellipsis, or double the length for /both.

```
(show #f (with ((ellipsis "...")) (trimmed 5 "abcde")))
=>  "abcde"

(show #f (with ((ellipses "...")) (trimmed 5 "abcdef")))
=>  "ab..."
```

It is an error if width is shorter than the width of the ellipsis.

(trimmed/lazy width fmt ...)

A variant of trimmed which generates each fmt in left to right order, and truncates and terminates immediately if more than width characters are generated. Thus this is safe to use with an infinite amount of output, e.g. from written-simply on an infinite list.

(fitted width fmt ...)

(fitted/right width fmt ...)

(fitted/both width fmt ...)

A combination of padded and trimmed that ensures that the output width is exactly width, truncating if it goes over and padding if it goes under.

(columnar column ...)

Formats each column side-by-side, i.e. as though each were formatted separately and then the individual lines concatenated together. The current line width (from the width state variable) is divided evenly among the columns, and all but the last column are right-padded. For example

```
(show #t (columnar (displayed "abc\ndef\n")
                   (displayed "123\n456\n")))
```

outputs

```
abc     123
def     456
```

assuming a 16-char width (the left side gets half the width, or 8 spaces, and is left aligned). Note that we explicitly use displayed instead of the strings directly. This is because columnar treats raw strings as literals inserted into the given location on every line, to be used as borders, for example:

```
(show #t (columnar "/* " (displayed "abc\ndef\n")
                   " | " (displayed "123\n456\n")
                   " */"))
```

would output

```
/* abc | 123 */
/* def | 456 */
```

You may also prefix any column with any of the symbols 'left, 'right or 'center to control the justification. The symbol 'infinite can be used to indicate the column generates an infinite stream of output.

You can further prefix any column with a width modifier. Any positive integer is treated as a fixed width, ignoring the available width. Any real number between 0 and 1 indicates a fraction of the available width (after subtracting out any fixed widths). Columns with unspecified width divide up the remaining width evenly. If the extra space does not divide evenly, it is allocated column-wise left to right, e.g. if the width of 78 is divided among 5 columns, the column widths become 16, 16, 16, 15, 15 in order.

Note that columnar builds its output incrementally, interleaving calls to the generators until each has produced a line, then concatenating that line together and outputting it. This is important because as noted above, some columns may produce an infinite stream of output, and in general you may want to format data larger than can fit into memory. Thus columnar would be suitable for line numbering a file of arbitrary size, or implementing the Unix yes(1) command, etc.

`(tabular column ...)`

Equivalent to columnar except that each column is padded at least to the minimum width required on any of its lines. Thus

```
(show #t (tabular "|" (each "a\nbc\ndef\n") "|"
                      (each "123\n45\n6\n") "|"))
```

outputs

```
|a  |123|
|bc |45 |
```

```
|def|6  |
```

This makes it easier to generate tables without knowing widths in advance. However, because it requires generating the entire output in advance to determine the correct column widths, tabular cannot format a table larger than would fit in memory. (wrapped fmt …)

Behaves like each, except text is accumulated and lines are wrapped to fit in the current width as in the Unix fmt(1) command. Specifically, words are tokenized by splitting on all characters which satisfy the predicate in the parameter word-separator?, which defaults to char-whitespace?. Words are grouped into lines separating them by space, and line breaks are introduced to minimize the sum of the cube of trailing whitespace on every line.

(wrapped/list list-of-strings)

Like wrapped, but taking a pre-tokenized list of strings.

(wrapped/char fmt ...)

As wrapped, but splits simply on individual characters exactly as the current width is reached on each line. Thus there is nothing to optimize and this formatter doesn't buffer output.

(justified <format> ...)

Like wrapped except the lines are full-justified.

```
(define func
  '(define (fold kons knil ls)
     (let lp ((ls ls) (acc knil))
       (if (null? ls) acc (lp (cdr ls) (kons (car ls) acc))))))

(define doc
  (string-append
    "The fundamental list iterator.  Applies KONS to each "
    "element of LS and the result of the previous application, "
    "beginning with KNIL.  With KONS as CONS and KNIL as '(), "
    "equivalent to REVERSE."))

(show #t (columnar (pretty func) " ; " (justified doc)))
```

outputs

```
(define (fold kons knil ls)          ; The  fundamental  list  iterator.
  (let lp ((ls ls) (acc knil))       ; Applies  KONS  to  each  element  of
    (if (null? ls)                   ; LS  and  the  result of the previous
        acc                          ; application,  beginning  with  KNIL.
```

```
        (lp (cdr ls)                    ; With  KONS  as CONS and KNIL as '(),
            (kons (car ls) acc)))))     ; equivalent to REVERSE.
```

(from-file pathname)

Displays the contents of the file pathname one line at a time, so that in typical formatters such as columnar only constant memory is consumed, making this suitable for formatting files of arbitrary size.

(line-numbers [start])

A convenience utility, just formats an infinite stream of numbers (in the current radix) beginning with start, which defaults to 1.

The Unix nl(1) utility could be implemented as:

```
(show #t (columnar 4 'right 'infinite (line-numbers)
                   " " (from-file "read-line.scm")))
```

which might output:

```
1
2 (define (read-line . o)
3   (let ((port (if (pair? o) (car o) (current-input-port))))
4     (let lp ((res '()))
5       (let ((c (read-char port)))
6         (if (or (eof-object? c) (eqv? c #\newline))
7             (list->string (reverse res))
8             (lp (cons c res)))))))
```

(as-red fmt ...)

(as-blue fmt ...)

(as-green fmt ...)

(as-cyan fmt ...)

(as-yellow fmt ...)

(as-magenta fmt ...)

(as-white fmt ...)

(as-black fmt ...)

(as-bold fmt ...)

(as-underline fmt ...)

Outputs the formatters colored or (boldened or underline) with ANSI escapes, for use when formatting to a terminal.

(as-unicode fmt ...)

Equivalent to

```
(with ((string-width unicode-terminal-width)) fmt ...)
```

Padding, trimming and tabbing, etc. will generally not do the right thing in the presence of zero-width and double-width Unicode characters. This formatter overrides the string-width state var used in column tracking to do the right thing in such cases, considering Unicode double or full width characters as 2 characters wide (as they typically are in fixed-width terminals), while treating combining and non-spacing characters as 0 characters wide.

```
;; 3 characters padded to 5
(show #f (with ((pad-char #\ )) (padded/both 5 "  ")))
=> "   "

;; the 3 characters have a terminal width of 6 so are not padded
(show #f (as-unicode (with ((pad-char #\ )) (padded/both 5 "  "))))
=> "  "
```

(unicode-terminal-width str)

A utility function which returns the integer number of columns str would require in a terminal, according to the following rules:

- non-spacing characters (format control characters with the property Cf, or non-spacing marks with the property Mn) count as 0 columns
- characters with the East Asian Wide (W) or East Asian Fullwidth (F) properties, according to Unicode TR #11, count as 2 columns
- characters with the Halfwidth (H) or Narrow (Na) should count as 1 column
- characters with the Neutral (N) non-East Asian also count as 1 column
- characters with the Ambiguous (A) property are implementation defined
- ANSI terminal escapes, as output by the color formatters above, count as 0 columns
- the tab character is implementation defined
- Implementations should support the properties from at least the current Unicode specification at time of writing this SRFI, 10.0.0. Higher Order Formatters and State

Formatters up to this point have been simple accumulators of output, with no control flow or handling of state. Both of these are provided by fn and with for getting and setting state, respectively.

A formatter is essentially an environment monad, although the underlying implementation is unspecified.

(fn ((id state-var) ...) expr ... fmt)

Short for "function," this is the analog to lambda. Returns a formatter which on application evaluates each expr and fmt in left-to-right order, in a lexical environment extended with each identifier id bound to the current value of the state variable named by the symbol state-var. The result of the fmt is then applied as a formatter.

As a convenience, any (id state-var) list may be abbreviated as simply id, indicating id is bound to the state variable of the same (symbol) name.

```
(show #f "column: " (fn (col) col))
=> "column: 8"

(show #f "column: " (fn ((col1 col))
                      (each col1 ", " (fn ((col2 col)) col2))))
=> "column: 8, 11"
```

The trivial case of no state variables is often useful to allow for lazy applications of formatters, needed for conditional formatting and loops. For example:

```
(show #t (let lp ((ls ls))
           (if (pair? ls)
               (each (car ls) (lp (cdr ls)))
               nothing)))
```

would eagerly create a formatter concatenating every element of ls before starting to accumulate any output, whereas

```
(show #t (let lp ((ls ls))
           (if (pair? ls)
               (each (car ls) (fn () (lp (cdr ls))))
               nothing)))
```

would lazily apply the formatters one at a time.

(with ((state-var value) ...) fmt ...)

Conceptually the formatting equivalent of parameterize, temporarily altering state variables. Applies each of the formatters fmt with each state-var bound to the corresponding value. The resulting state is then updated to restore each state-var to its original value.

`(with! (state-var value) ...)`

Similar to with but does not restore the original values, changing the value of each state-var for any remaining formatters in a sequence.

`(forked fmt1 fmt2)`

Calls fmt1 on (a conceptual copy of) the current state, then fmt2 on the same original state as though fmt1 had not been called.

`(call-with-output formatter mapper)`

A utility, calls formatter on a copy of the current state (as with forked), accumulating the results into a string. Then calls the formatter resulting from (mapper result-string) on the original state.

`port`

The textual port output is written to, this can be overridden to capture intermediate output.

`row`

The current row of output.

`col`

The current column of output, used for padding and spacing, etc.

`width`

The current line width, used for wrapping, pretty-printing, and columnar formatting. The default is implementation-defined.

`output`

The underlying standard formatter for writing a single string. The default value outputs the string while tracking the current row and col. This can be overridden both to capture intermediate output and perform transformations on strings before outputting, but should generally wrap the existing output to preserve expected behavior.

`writer`

The mapper for automatic formatting of non-string/char values in top-level show, each and other formatters. Default value is implementation-defined.

string-width

A function of a single string, it returns the length in columns of that string, used by the default output.

pad-char

The character used by space-to, tab-to and other padding formatters.

```
(define (print-table-of-contents alist)
  (define (print-line x)
    (each (car x) (space-to 72) (padded 3 (cdr x))))
  (show #t (with ((pad-char #\.))
             (joined/suffix print-line alist nl))))

(print-table-of-contents
 '(("An Unexpected Party" . 29)
   ("Roast Mutton" . 60)
   ("A Short Rest" . 87)
   ("Over Hill and Under Hill" . 100)
   ("Riddles in the Dark" . 115)))
```

would output

```
An Unexpected Party.....................................................29
Roast Mutton............................................................60
A Short Rest............................................................87
Over Hill and Under Hill...............................................100
Riddles in the Dark.....................................................115
```

eellipsis

The string used when truncating as described in trimmed.

radix

The radix for numeric output, defaulting to 10, as used in numeric and written.

precision

The precision for numeric output, as described in numeric and written. The precision specifies the number of digits written after the decimal point. If the numeric value to be written out requires more digits to represent it than precision, the written representation is chosen which is closest to the numeric value and representable with the specified precision. If the numeric value falls on the midpoint of two such representations, it is implementation dependent which representation is chosen.

When the numeric value is an inexact floating-point number, there is more than one interpretation of this "rounding". One is to take the effective value the floating-point number represents (e.g. if we use binary floating-point numbers, we take the value of (* sign mantissa (expt 2 exponent))), and compare it to the two closest numeric representations of the given precision. Another way is to obtain the default notation of the floating-point number and apply rounding to it. The former (we call it effective rounding) is consistent with most floating-point number operations, but may lead to a more non-intuitive result than the latter (we call it notational rounding). For example, 5.015 can't be represented exactly in binary floating-point numbers. With IEEE754 floating-point numbers, the floating point number closest to 5.015 is smaller than exact 5.015, i.e. (< 5.015 5015/1000) => #t. With effective rounding with precision 2, it should result in "5.01". However, users who look at the notation may be confused by "5.015" not being rounded up as they usually expect. With notational rounding the implementation chooses "5.02" (if it also adopts round-half-to-infinity or round-half-up rule). It is up to the implementation to choose which interpretation to adopt.

decimal-sep

The decimal separator for floating point output, default ".".

decimal-align

Specifies an alignment for the decimal place when formatting numbers, and is useful for outputting tables of numbers.

```
(define (print-angles x)
  (joined numeric (list x (sin x) (cos x) (tan x)) " "))

(show #t (with ((decimal-align 5) (precision 3))
           (joined/suffix print-angles (iota 5) nl)))
```

would output

```
 0.000    0.000    1.000    0.000
 1.000    0.842    0.540    1.557
 2.000    0.909   -0.416   -2.185
 3.000    0.141   -0.990   -0.142
 4.000   -0.757   -0.654    1.158
```

word-separator?

A character predicate used to tokenize words for wrapped and justify. Defaults to char-whitespace?. More flexibility is available with wrapped/list. # `(scheme vector)`

This library is based on SRFI-133.

(make-vector size [fill])

[R7RS-small] Creates and returns a vector of size size. If fill is specified, all the elements of the vector are initialized to fill. Otherwise, their contents are indeterminate.

Example:

```
(make-vector 5 3)
#(3 3 3 3 3)
```

(vector x ...)

[R7RS-small] Creates and returns a vector whose elements are x ….

Example:

```
(vector 0 1 2 3 4)
#(0 1 2 3 4)
```

(vector-unfold f length initial-seed ...)

The fundamental vector constructor. Creates a vector whose length is length and iterates across each index k between 0 and length, applying f at each iteration to the current index and current seeds, in that order, to receive n + 1 values: first, the element to put in the kth slot of the new vector and n new seeds for the next iteration. It is an error for the number of seeds to vary between iterations. Note that the termination condition is different from the unfold procedure of SRFI 1.

Examples:

“scheme (vector-unfold ((i x) (values x (- x 1))) 10 0) #(0 -1 -2 -3 -4 -5 -6 -7 -8 -9)

```
Construct a vector of the sequence of integers in the range [0,n).

```scheme
(vector-unfold values n)
#(0 1 2 ... n-2 n-1)
```

Copy vector.

```
    (vector-unfold ( (i) (vector-ref vector i))
                   (vector-length vector))
```

(vector-unfold-right f length initial-seed ...)

Like vector-unfold, but it uses f to generate elements from right-to-left, rather than left-to-right. The first index used is length - 1. Note that the termination condition is different from the unfold-right procedure of SRFI 1.
```

Examples:

Construct a vector of pairs of non-negative integers whose values sum to 4.

```
(vector-unfold-right (  (i x) (values (cons i x) (+ x 1))) 5 0)
#((0 . 4) (1 . 3) (2 . 2) (3 . 1) (4 . 0))
```

Reverse vector.

```
(vector-unfold-right (  (i x) (values (vector-ref vector x) (+ x 1)))
                      (vector-length vector)
                      0)
```

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

[R7RS-small] Allocates a new vector whose length is end - start and fills it with elements from vec, taking elements from vec starting at index start and stopping at index end. Start defaults to 0 and end defaults to the value of (vector-length vec). SRFI 43 provides an optional fill argument to supply values if end is greater than the length of vec. Neither R7RS-small nor this SRFI requires support for this argument.

Examples:

```
(vector-copy '#(a b c d e f g h i))
#(a b c d e f g h i)

(vector-copy '#(a b c d e f g h i) 6)
#(g h i)

(vector-copy '#(a b c d e f g h i) 3 6)
#(d e f)
```

(vector-reverse-copy vec [start [end]])

Like vector-copy, but it copies the elements in the reverse order from vec.

Example:

```
(vector-reverse-copy '#(5 4 3 2 1 0) 1 5)
#(1 2 3 4)
```

(vector-append vec ...)

[R7RS-small] Returns a newly allocated vector that contains all elements in order from the subsequent locations in vec ....

Examples:

```
(vector-append '#(x) '#(y))
#(x y)
```

```
(vector-append '#(a) '#(b c d))
#(a b c d)

(vector-append '#(a #(b)) '#(#(c)))
#(a #(b) #(c))
```

(vector-concatenate list-of-vectors)

Appends each vector in list-of-vectors. This is equivalent to:

```
(apply vector-append list-of-vectors)
```

However, it may be implemented better.

Example:

```
(vector-concatenate '(#(a b) #(c d)))
#(a b c d)
```

(vector-append-subvectors [vec start end] ...)

Returns a vector that contains every element of each vec from start to end in the specified order. This procedure is a generalization of vector-append.

Example:

```
(vector-append-subvectors '#(a b c d e) 0 2 '#(f g h i j) 2 4)
#(a b h i)
```

(vector? x)

[R7RS-small] Disjoint type predicate for vectors: this returns #t if x is a vector, and #f if otherwise.

Examples:

```
(vector? '#(a b c))
#t

(vector? '(a b c))
#f

(vector? #t)
#f

(vector? '#())
#t
```

```
(vector? '())
#f
```

(vector-empty? vec)

Returns #t if vec is empty, i.e. its length is 0, and #f if not.

Examples:

```
(vector-empty? '#(a))
#f

(vector-empty? '#(()))
#f

(vector-empty? '#(#()))
#f

(vector-empty? '#())
#t
```

(vector= elt=? vec ...)

Vector structure comparator, generalized across user-specified element comparators. Vectors a and b are considered equal by vector= iff their lengths are the same, and for each respective element Ea and Eb, (elt=? Ea Eb) returns a true value. Elt=? is always applied to two arguments.

If there are only zero or one vector arguments, #t is automatically returned. The dynamic order in which comparisons of elements and of vectors are performed is left completely unspecified; do not rely on a particular order.

Examples:

```
(vector= eq? '#(a b c d) '#(a b c d))
#t

(vector= eq? '#(a b c d) '#(a b d c))
#f

(vector= = '#(1 2 3 4 5) '#(1 2 3 4))
#f

(vector= = '#(1 2 3 4) '#(1 2 3 4))
#t

The two trivial cases.
```

```
(vector= eq?)
#t

(vector= eq? '#(a))
#t

Note the fact that we don't use vector literals in the next two - it is unspecified whether

(vector= eq? (vector (vector 'a)) (vector (vector 'a)))
#f

(vector= equal? (vector (vector 'a)) (vector (vector 'a)))
#t
```

(vector-ref vec i)

[R7RS-small] Vector element dereferencing: returns the value that the location in vec at i is mapped to in the store. Indexing is based on zero. I must be within the range [0, (vector-length vec)).

Example:

```
(vector-ref '#(a b c d) 2)
c
```

(vector-length vec)

[R7RS-small] Returns the length of vec, the number of locations reachable from vec. (The careful word 'reachable' is used to allow for 'vector slices,' whereby vec refers to a larger vector that contains more locations that are unreachable from vec. This SRFI does not define vector slices, but later SRFIs may.)

Example:

```
(vector-length '#(a b c))
3
```

(vector-fold kons knil vec1 vec2 ...)

The fundamental vector iterator. Kons is iterated over each value in all of the vectors, stopping at the end of the shortest; kons is applied as (kons state (vector-ref vec1 i) (vector-ref vec2 i) ...) where state is the current state value — the current state value begins with knil, and becomes whatever kons returned on the previous iteration —, and i is the current index.

The iteration is strictly left-to-right.

Examples:

Find the longest string's length in vector-of-strings.

```
(vector-fold (  (len str) (max (string-length str) len))
               0 vector-of-strings)
```

Produce a list of the reversed elements of vec.

```
(vector-fold (  (tail elt) (cons elt tail))
               '() vec)
```

Count the number of even numbers in vec.

```
(vector-fold (  (counter n)
                 (if (even? n) (+ counter 1) counter))
               0 vec)
```

(vector-fold-right kons knil vec1 vec2 ...)

Similar to vector-fold, but it iterates right to left instead of left to right.

Example:

Convert a vector to a list.

```
(vector-fold-right (  (tail elt) (cons elt tail))
                     '() '#(a b c d))
(a b c d)
```

(vector-map f vec1 vec2 ...)

[R7RS-small] Constructs a new vector of the shortest size of the vector arguments. Each element at index i of the new vector is mapped from the old vectors by (f (vector-ref vec1 i) (vector-ref vec2 i) …). The dynamic order of application of f is unspecified.

Examples:

```
(vector-map (  (x) (* x x))
              (vector-unfold (  (i x) (values x (+ x 1))) 4 1))
#(1 4 9 16)

(vector-map (  (x y) (* x y))
              (vector-unfold (  (x) (values x (+ x 1))) 5 1)
              (vector-unfold (  (x) (values x (- x 1))) 5 5))
#(5 8 9 8 5)

(let ((count 0))
   (vector-map (  (ignored-elt)
                  (set! count (+ count 1))
                  count)
               '#(a b)))
#(1 2) OR #(2 1)
```

(vector-map! f vec1 vec2 ...)

Similar to vector-map, but rather than mapping the new elements into a new vector, the new mapped elements are destructively inserted into vec1. Again, the dynamic order of application of f unspecified, so it is dangerous for f to apply either vector-ref or vector-set! to vec1 in f.

(vector-for-each f vec1 vec2 ...)

[R7RS-small] Simple vector iterator: applies f to the corresponding list of parallel elements from vec1 vec2 … in the range [0, length), where length is the length of the smallest vector argument passed, In contrast with vector-map, f is reliably applied to each subsequent element, starting at index 0, in the vectors.

Example:

```
    (vector-for-each (  (x) (display x) (newline))
                     '#("foo" "bar" "baz" "quux" "zot"))
```

Displays:

```
foo
bar
baz
quux
zot
```

(vector-count pred? vec1 vec2 ...)

Counts the number of parallel elements in the vectors that satisfy pred?, which is applied, for each index i in the range [0, length) where length is the length of the smallest vector argument, to each parallel element in the vectors, in order.

Examples:

```
(vector-count even? '#(3 1 4 1 5 9 2 5 6))
3

(vector-count < '#(1 3 6 9) '#(2 4 6 8 10 12))
2
```

(vector-cumulate f knil vec)

Returns a newly allocated vector new with the same length as vec. Each element i of new is set to the result of invoking f on newi-1 and veci, except that for the first call on f, the first argument is knil. The new vector is returned.

Note that the order of arguments to vector-cumulate was changed by errata-3 on 2016-09-02.

Example:

```
(vector-cumulate + 0 '#(3 1 4 1 5 9 2 5 6))
#(3 4 8 9 14 23 25 30 36)
```

(vector-index pred? vec1 vec2 ...)

Finds & returns the index of the first elements in vec1 vec2 … that satisfy pred?. If no matching element is found by the end of the shortest vector, #f is returned.

Examples:

```
(vector-index even? '#(3 1 4 1 5 9))
2

(vector-index < '#(3 1 4 1 5 9 2 5 6) '#(2 7 1 8 2))
1

(vector-index = '#(3 1 4 1 5 9 2 5 6) '#(2 7 1 8 2))
#f
```

(vector-index-right pred? vec1 vec2 ...)

Like vector-index, but it searches right-to-left, rather than left-to-right, and all of the vectors must have the same length.

(vector-skip pred? vec1 vec2 ...)

Finds & returns the index of the first elements in vec1 vec2 … that do not satisfy pred?. If all the values in the vectors satisfy pred? until the end of the shortest vector, this returns #f. This is equivalent to:

“scheme (vector-index ((x1 x2 …) (not (pred? x1 x1 …))) vec1 vec2 …)

Example:

```scheme
(vector-skip number? '#(1 2 a b 3 4 c d))
2
```

(vector-skip-right pred? vec1 vec2 ...)

Like vector-skip, but it searches for a non-matching element right-to-left, rather than left-to-right, and it is an error if all of the vectors do not have the same length. This is equivalent to:

```
(vector-index-right ( (x1 x2 ...) (not (pred? x1 x1 ...)))
                    vec1 vec2 ...)
```

(vector-binary-search vec value cmp)

Similar to vector-index and vector-index-right, but instead of searching left to right or right to left, this performs a binary search. If there is more than one element of vec that matches value in the sense of cmp, vector-binary-search may return the index of any of them.

cmp should be a procedure of two arguments and return a negative integer, which indicates that its first argument is less than its second, zero, which indicates that they are equal, or a positive integer, which indicates that the first argument is greater than the second argument. An example cmp might be:

```
(  (char1 char2)
  (cond ((char<? char1 char2) -1)
        ((char=? char1 char2) 0)
        (else 1)))
```

(vector-any pred? vec1 vec2 ...)

Finds the first set of elements in parallel from vec1 vec2 … for which pred? returns a true value. If such a parallel set of elements exists, vector-any returns the value that pred? returned for that set of elements. The iteration is strictly left-to-right.

(vector-every pred? vec1 vec2 ...)

If, for every index i between 0 and the length of the shortest vector argument, the set of elements (vector-ref vec1 i) (vector-ref vec2 i) … satisfies pred?, vector-every returns the value that pred? returned for the last set of elements, at the last index of the shortest vector. The iteration is strictly left-to-right.

(vector-partition pred? vec)

A vector the same size as vec is newly allocated and filled with all the elements of vec that satisfy pred? in their original order followed by all the elements that do not satisfy pred?, also in their original order.

Two values are returned, the newly allocated vector and the index of the leftmost element that does not satisfy pred?.

(vector-set! vec i value)

[R7RS-small] Assigns the contents of the location at i in vec to value.

(vector-swap! vec i j)

Swaps or exchanges the values of the locations in vec at i & j.

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

[R7RS-small] Assigns the value of every location in vec between start, which defaults to 0 and end, which defaults to the length of vec, to fill.

`(vector-reverse! vec [start [end]])`

Destructively reverses the contents of the sequence of locations in vec between start and end. Start defaults to 0 and end defaults to the length of vec. Note that this does not deeply reverse.

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

[R7RS-small] 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-reverse-copy! to at from [start [end]])`

Like vector-copy!, but the elements appear in to in reverse order.

`(vector-unfold! f vec start end initial-seed ...)`

Like vector-unfold, but the elements are copied into the vector vec starting at element start rather than into a newly allocated vector. Terminates when end-start elements have been generated.

`(vector-unfold-right! f vec start end initial-seed ...)`

Like vector-unfold!, but the elements are copied in reverse order into the vector vec starting at the index preceding end.

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

[R7RS-small] Creates a list containing the elements in vec between start, which defaults to 0, and end, which defaults to the length of vec.

`(reverse-vector->list vec [start [end]])`

Like vector->list, but the resulting list contains the elements in reverse of vec.

`(list->vector proper-list)`

[R7RS-small] Creates a vector of elements from proper-list.

`(reverse-list->vector proper-list)`

Like list->vector, but the resulting vector contains the elements in reverse of proper-list.

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

[R7RS-small] Creates a vector containing the elements in string between start, which defaults to 0, and end, which defaults to the length of string.

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

[R7RS-small] Creates a string containing the elements in vec between start, which defaults to 0, and end, which defaults to the length of vec. It is an error if the elements are not characters. # `(scheme r5rs)`

Thie library exportR5RS 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 bitwise)

This library is based on SRFI-151.

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

(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 #x10000000000000000000000000000000 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)))
```

(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 rlist)`

This library is based on SRFI-101.

Random-access lists [1] are a purely functional data structure for representing lists of values. A random-access list may act as a drop in replacement for the usual linear-access pair and list data structures (pair?, cons, car, cdr), which additionally supports fast index-based addressing and updating (list-ref, list-set). The impact is a whole class of purely-functional algorithms expressed in terms

of index-based list addressing become feasible compared with their linear-access list counterparts.

This document proposes a library API for purely functional random-access lists consistent with the R6RS [2] base library and list utility standard library [3]. # `(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 `(arew 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 box)`

This library is based on SRFI-111.

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

`(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.

`(scheme lazy)`

`(delay exp)`

TODO

`(force promise)`

TODO

`(delay-force exp)`

TODO

`(promise? obj)`

TODO

`(make-promise exp)`

TODO # `(scheme regex)`

This library is based on SRFI-115.

This library provides a library for matching strings with regular expressions described using the SRE "Scheme Regular Expression" notation first introduced by SCSH, and extended heavily by IrRegex.

`(regexp re)`

Compiles a regexp if given an object whose structure matches the SRE syntax. This may be written as a literal or partial literal with quote or quasiquote, or may be generated entirely programmatically. Returns re unmodified if it is already a regexp. Raises an error if re is neither a regexp nor a valid representation of an SRE.

Mutating re may invalidate the resulting regexp, causing unspecified results if subsequently used for matching.

`(rx sre ...)`

Macro shorthand for (regexp '(: sre …)). May be able to perform some or all computation at compile time if sre is not unquoted. Note because of this equivalence with the procedural constructor regexp, the semantics of unquote differs from the original SCSH implementation in that unquoted expressions can expand into any object matching the SRE syntax, but not a compiled regexp object. Further, unquote and unquote-splicing both expand all matches.

Rationale: Providing a procedural interface provides for greater flexibility, and without loss of potential compile-time optimizations by preserving the syntactic shorthand. The alternative is to rely on eval to dynamically generate regular expressions. However regexps in many cases come from untrusted sources, such as search parameters to a server, or from serialized sources such as config files or command-line arguments. Moreover many applications may want to keep many thousands of regexps in memory at once. Given the relatively heavy cost and

insecurity of eval, and the frequency with which SREs are read and written as text, we prefer the procedural interface.

`(regexp->sre re)`

Returns an SRE corresponding to the given regexp re. The SRE will be equivalent to (will match the same strings) but not necessarily equal? to the SRE originally used to compile re. Mutating the result may invalidate re, causing unspecified results if subsequently used for matching.

`(char-set->sre char-set)`

Returns an SRE corresponding to the given SRFI 14 character set. The resulting SRE expands the character set into notation which does not make use of embedded SRFI 14 character sets, and so is suitable for writing portably.

`(valid-sre? obj)`

Returns true iff obj can be safely passed to regexp.

`(regexp? obj)`

Returns true iff obj is a regexp.

`(regexp-matches re str [start [end]])`

Returns an regexp-match object if re successfully matches the entire string str from start (inclusive) to end (exclusive), or #f is the match fails. The regexp-match object will contain information needed to extract any submatches.

`(regexp-matches? re str [start [end]])`

Returns #t if re matches str as in regexp-matches, or #f otherwise. May be faster than regexp-matches since it doesn't need to return submatch data.

`(regexp-search re str [start [end]])`

Returns a regexp-match object if re successfully matches a substring of str between start (inclusive) and end (exclusive), or #f if the match fails. The regexp-match object will contain information needed to extract any submatches.

`(regexp-fold re kons knil str [finish [start [end]]])`

The fundamental regexp matching iterator. Repeatedly searches str for the regexp re so long as a match can be found. On each successful match, applies (kons i regexp-match str acc) where i is the index since the last match (beginning with start), regexp-match is the resulting match, and acc is the result of the

previous kons application, beginning with knil. When no more matches can be found, calls finish with the same arguments, except that regexp-match is #f.

By default finish just returns acc.

```
(regexp-fold 'word
             (lambda (i m str acc)
               (let ((s (regexp-match-submatch m 0)))
                (cond ((assoc s acc)
                       => (lambda (x) (set-cdr! x (+ 1 (cdr x))) acc))
                      (else `((,s . 1) ,@acc)))))
             '()
             "to be or not to be")
=> '(("not" . 1) ("or" . 1) ("be" . 2) ("to" . 2))
```

(regexp-extract re str [start [end]])

Extracts all non-empty substrings of str which match re between start and end as a list of strings.

```
(regexp-extract '(+ numeric) "192.168.0.1")
=> ("192" "168" "0" "1")
```

(regexp-split re str [start [end]])

Splits str into a list of (possibly empty) strings separated by non-empty matches of re.

```
(regexp-split '(+ space) " fee fi  fo\tfum\n")
=> ("" "fee" "fi" "fo" "fum" "")
(regexp-split '(",;") "a,,b,")
=> ("a" "" "b" "")
(regexp-split '(* digit) "abc123def456ghi789")
=> ("abc" "def" "ghi" "")
```

(regexp-partition re str [start [end]])

Partitions str into a list of non-empty strings matching re, interspersed with the unmatched portions of the string. The first and every odd element is an unmatched substring, which will be the empty string if re matches at the beginning of the string or end of the previous match. The second and every even element will be a substring matching re. If the final match ends at the end of the string, no trailing empty string will be included. Thus, in the degenerate case where str is the empty string, the result is (“ “).

Note that regexp-partition is equivalent to interleaving the results of regexp-split and regexp-extract, starting with the former.

```
(regexp-partition '(+ (or space punct)) "")
=> ("")
(regexp-partition '(+ (or space punct)) "Hello, world!\n")
=> ("Hello" ", " "world" "!\n")
(regexp-partition '(+ (or space punct)) "¿Dónde Estás?")
=> ("" "¿" "Dónde" " " "Estás" "?")
(regexp-partition '(* digit) "abc123def456ghi789")
=> ("abc" "123" "def" "456" "ghi" "789")
```

(regexp-replace re str subst [start [end [count]]])

Returns a new string replacing the countth match of re in str the subst, where the zero-indexed count defaults to zero (i.e. the first match). If there are not count matches, returns the selected substring unmodified.

subst can be a string, an integer or symbol indicating the contents of a numbered or named submatch of re, 'pre for the substring to the left of the match, or 'post for the substring to the right of the match.

The optional parameters start and end restrict both the matching and the substitution, to the given indices, such that the result is equivalent to omitting these parameters and replacing on (substring str start end). As a convenience, a value of #f for end is equivalent to (string-length str).

```
(regexp-replace '(+ space) "one two three" "_")
=> "one_two three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 0)
=> "one_two three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 1)
=> "one two_three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 2)
=> "one two three"
```

(regexp-replace-all re str subst [start [end]])

Equivalent to regexp-replace, but replaces all occurrences of re in str.

```
(regexp-replace-all '(+ space) "one two three" "_")
=> "one_two_three"
```

(regexp-match? obj)

Returns true iff obj is a successful match from regexp-matches or regexp-search.

```
(regexp-match? (regexp-matches "x" "x"))  => #t
(regexp-match? (regexp-matches "x" "y"))  => #f
```

(regexp-match-count regexp-match)

Returns the number of submatches of regexp-match, regardless of whether they matched or not. Does not include the implicit zero full match in the count.

```
(regexp-match-count (regexp-matches "x" "x"))  => 0
(regexp-match-count (regexp-matches '($ "x") "x"))  => 1
```

(regexp-match-submatch regexp-match field)

Returns the substring matched in regexp-match corresponding to field, either an integer or a symbol for a named submatch. Index 0 refers to the entire match, index 1 to the first lexicographic submatch, and so on. If there are multiple submatches with the same name, the first which matched is returned. If passed an integer outside the range of matches, or a symbol which does not correspond to a named submatch of the pattern, it is an error. If the corresponding submatch did not match, returns false.

The result of extracting a submatch after the original matched string has been mutated is unspecified.

```
(regexp-match-submatch (regexp-search 'word "**foo**") 0)  => "foo"
(regexp-match-submatch
  (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => "*foo*"
(regexp-match-submatch
  (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => "foo"
```

(regexp-match-submatch-start regexp-match field)

Returns the start index regexp-match corresponding to field, as in regexp-match-submatch.

```
(regexp-match-submatch-start
 (regexp-search 'word "**foo**") 0)  => 2
(regexp-match-submatch-start
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => 1
(regexp-match-submatch-start
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => 2
```

(regexp-match-submatch-end regexp-match field)

Returns the end index in regexp-match corresponding to field, as in regexp-match-submatch.

```
(regexp-match-submatch-end
 (regexp-search 'word "**foo**") 0)  => 5
(regexp-match-submatch-end
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => 6
```

```
(regexp-match-submatch-end
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => 5
```

(regexp-match->list regexp-match)

Returns a list of all submatches in regexp-match as string or false, beginning with the entire match 0.

```
(regexp-match->list
 (regexp-search '(: ($ word) (+ (or space punct)) ($ word)) "cats & dogs"))
 => '("cats & dogs" "cats" "dogs")
```

SRE Syntax

The grammar for SREs is summarized below. Note that an SRE is a first-class object consisting of nested lists of strings, chars, char-sets, symbols and numbers. Where the syntax is described as (foo bar), this can be constructed equivalently as '(foo bar) or (list 'foo 'bar), etc. The following sections explain the semantics in greater detail.

"'scheme ::= | ; A literal string match. | ; A character set match. | (* ...) ; 0 or more matches. | (zero-or-more ...) | (+ ...) ; 1 or more matches. | (one-or-more ...) | (? ...) ; 0 or 1 matches. | (optional ...) | (= ...) ; matches. | (exactly ...) | (>= ...) ; or more matches. | (at-least ...) | (** ...) ; to matches. | (repeated ...)

```
 | (|  <sre> ...)              ; Alternation.
 | (or <sre> ...)

 | (:   <sre> ...)             ; Sequence.
 | (seq <sre> ...)
 | ($ <sre> ...)               ; Numbered submatch.
 | (submatch <sre> ...)
 | (-> <name> <sre> ...)               ;  Named submatch.  <name> is
 | (submatch-named <name> <sre> ...)   ;  a symbol.

 | (w/case   <sre> ...)        ; Introduce a case-sensitive context.
 | (w/nocase <sre> ...)        ; Introduce a case-insensitive context.

 | (w/unicode   <sre> ...)     ; Introduce a unicode context.
 | (w/ascii <sre> ...)         ; Introduce an ascii context.

 | (w/nocapture <sre> ...)     ; Ignore all enclosed submatches.

 | bos                         ; Beginning of string.
 | eos                         ; End of string.

 | bol                         ; Beginning of line.
```

```
 | eol                          ; End of line.

 | bog                          ; Beginning of grapheme cluster.
 | eog                          ; End of grapheme cluster.
 | grapheme                     ; A single grapheme cluster.

 | bow                          ; Beginning of word.
 | eow                          ; End of word.
 | nwb                          ; A non-word boundary.
 | (word <sre> ...)            ; An SRE wrapped in word boundaries.
 | (word+ <cset-sre> ...)      ; A single word restricted to a cset.
 | word                         ; A single word.

 | (?? <sre> ...)              ; A non-greedy pattern, 0 or 1 match.
 | (non-greedy-optional <sre> ...)
 | (*? <sre> ...)              ; Non-greedy 0 or more matches.
 | (non-greedy-zero-or-more <sre> ...)
 | (**? <m> <n> <sre> ...)     ; Non-greedy <m> to <n> matches.
 | (non-greedy-repeated <sre> ...)
 | (look-ahead <sre> ...)      ; Zero-width look-ahead assertion.
 | (look-behind <sre> ...)     ; Zero-width look-behind assertion.
 | (neg-look-ahead <sre> ...)  ; Zero-width negative look-ahead assertion.
 | (neg-look-behind <sre> ...) ; Zero-width negative look-behind assertion.

 | (backref <n-or-name>)       ; Match a previous submatch.
```

The grammar for cset-sre is as follows.

```scheme
    <cset-sre> ::=
     | <char>                      ; literal char
     | "<char>"                    ; string of one char
     | <char-set>                  ; embedded SRFI 14 char set
     | (<string>)                  ; literal char set
     | (char-set <string>)
     | (/ <range-spec> ...)        ; ranges
     | (char-range <range-spec> ...)
     | (or <cset-sre> ...)         ; union
     | (|\|| <cset-sre> ...)
     | (and <cset-sre> ...)        ; intersection
     | (& <cset-sre> ...)
     | (- <cset-sre> ...)          ; difference
     | (- <difference> ...)
     | (~ <cset-sre> ...)          ; complement of union
     | (complement <cset-sre> ...)
     | (w/case <cset-sre> ...)     ; case and unicode toggling
```

```
  | (w/nocase <cset-sre> ...)
  | (w/ascii <cset-sre> ...)
  | (w/unicode <cset-sre> ...)
  | any | nonl | ascii | lower-case | lower
  | upper-case | upper | title-case | title
  | alphabetic | alpha | alphanumeric | alphanum | alnum
  | numeric | num | punctuation | punct | symbol
  | graphic | graph | whitespace | white | space
  | printing | print | control | cntrl | hex-digit | xdigit

<range-spec> ::= <string> | <char>
```

<string>

A literal string.

```
(regexp-search "needle" "hayneedlehay") => #<regexp-match>
(regexp-search "needle" "haynEEdlehay") => #f
```

(seq sre ...)

(: sre ...)

Sequencing. Matches if each of sre matches adjacently in order.

```
(regexp-search '(: "one" space "two" space "three") "one two three") => #<regexp-match>
```

(or sre ...)

(|\|| sre ...)

Alternation. Matches if any of sre match.

```
(regexp-search '(or "eeney" "meeney" "miney") "meeney") => #<regexp-match>
(regexp-search '(or "eeney" "meeney" "miney") "moe") => #f
```

(w/nocase sre ...)

Enclosed sres are case-insensitive. In a Unicode context character and string literals match with the default simple Unicode case-insensitive matching. Implementations may, but are not required to, handle variable length case conversions, such as #00DF "ß" matching the two characters "SS".

Character sets match if any character in the set matches case-insensitively to the input. Conceptually each cset-sre is expanded to contain all case variants for all of its characters. In a compound cset-sre the expansion is applied at the terminals consisting of characters, strings, embedded SRFI 14 char-sets, and named character sets. For simple unions this would be equivalent to computing the full union first and then expanding case variants, but the semantics can

differ when differences and intersections are applied. For example, (w/nocase (~ ("Aab"))) is equivalent to (~ ("AaBb")), for which "B" is clearly not a member. However if you were to compute (~ ("Aab")) first then you would have a char-set containing "B", and after expanding case variants both "B" and "b" would be members.

In an ASCII context only the 52 ASCII letters (/ "a-zA-Z") match case-insensitively to each other.

In a Unicode context the only named cset-sre which are affected by w/nocase are upper and lower. Note that the case insensitive versions of these are not equivalent to alpha as there are characters with the letter property but no case.

```
(regexp-search "needle" "haynEEdlehay") => #f
(regexp-search '(w/nocase "needle") "haynEEdlehay") => #<regexp-match>

(regexp-search '(~ ("Aab")) "B") => #<regexp-match>
(regexp-search '(~ ("Aab")) "b") => #f
(regexp-search '(w/nocase (~ ("Aab"))) "B") => #f
(regexp-search '(w/nocase (~ ("Aab"))) "b") => #f
(regexp-search '(~ (w/nocase ("Aab"))) "B") => #f
(regexp-search '(~ (w/nocase ("Aab"))) "b") => #f
```

(w/case sre ...)

Enclosed sres are case-sensitive. This is the default, and overrides any enclosing w/nocase setting.

```
(regexp-search '(w/nocase "SMALL" (w/case "BIG")) "smallBIGsmall") => #<regexp-match>
(regexp-search '(w/nocase (~ (w/case ("Aab")))) "b") => #f
```

(w/ascii sre ...)

Enclosed sres are interpreted in an ASCII context. In practice many regular expressions are used for simple parsing and only ASCII characters are relevant. Switching to ASCII mode can improve performance in some implementations.

```
(regexp-search '(w/ascii bos (* alpha) eos) "English") => #<regexp-match>
(regexp-search '(w/ascii bos (* alpha) eos) "E    ") => #f
```

(w/unicode sre ...)

Enclosed sres are interpreted in a Unicode context - character sets with both an ASCII and Unicode definition take the latter. Has no effect if the regexp-unicode feature is not provided. This is the default.

```
(regexp-search '(w/unicode bos (* alpha) eos) "English") => #<regexp-match>
(regexp-search '(w/unicode bos (* alpha) eos) "E    ") => #<regexp-match>
```

(w/nocapture sre ...)

Disables capturing for all submatches ($, submatch, -> and submatch-named) in the enclosed sres. The resulting SRE matches exactly the same strings, but without any associated submatch info. Useful for utility SREs which you want to incorporate without affecting your submatch positions.

```
(let ((number '($ (+ digit))))
  (cdr
   (regexp-match->list
    (regexp-search `(: ,number "-" ,number "-" ,number)
                   "555-867-5309")))  ; => '("555" "867" "5309")
  (cdr
   (regexp-match->list
    (regexp-search `(: ,number "-" (w/nocapture ,number) "-" ,number)
                   "555-867-5309"))))   => '("555" "5309")
```

(optional sre ...)

(? sre ...)

An optional pattern - matches 1 or 0 times.

```
(regexp-search '(: "match" (? "es") "!") "matches!") => #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "match!") => #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "matche!") => #f
```

(zero-or-more sre ...)

(* sre ...)

Kleene star, matches 0 or more times.

```
(regexp-search '(: "<" (* (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (* (~ #\>)) ">") "<>") => #<regexp-match>
(regexp-search '(: "<" (* (~ #\>)) ">") "<html") => #f
```

(one-or-more sre ...)

(+ sre ...)

1 or more matches. Like * but requires at least a single match.

```
(regexp-search '(: "<" (+ (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (+ (~ #\>)) ">") "<a>") => #<regexp-match>
(regexp-search '(: "<" (+ (~ #\>)) ">") "<>") => #f
```

(at-least n sre ...)

(>= n sre ...)

More generally, n or more matches.

```
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<table>") => #<regexp-match>
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<pre>") => #<regexp-match>
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<tr>") => #f
```

(exactly n sre ...)

(= n sre ...)

Exactly n matches.

```
(regexp-search '(: "<" (= 4 (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (= 4 (~ #\>)) ">") "<table>") => #f
```

(repeated from to sre ...)

(** from to sre ...)

The most general form, from n to m matches, inclusive.

```
(regexp-search '(: (= 3 (** 1 3 numeric) ".") (** 1 3 numeric)) "192.168.1.10") => #<regexp-
(regexp-search '(: (= 3 (** 1 3 numeric) ".") (** 1 3 numeric)) "192.0168.1.10") => #f
```

(submatch sre ...)

($ sre ...)

A numbered submatch. The contents matching the pattern will be available in the resulting regexp-match.

(submatch-named name sre ...)

(-> name sre ...)

A named submatch. Behaves just like submatch, but the field may also be referred to by name.

(backref n-or-name)

Optional: Match a previously matched submatch. The feature regexp-backrefs will be provided if this pattern is supported. Backreferences are expensive, and can trivially be shown to be NP-hard, so one should avoid their use even in implementations which support them.

<char>

A singleton char set.

```
(regexp-matches '(* #\-) "---") => #<regexp-match>
(regexp-matches '(* #\-) "-_-") => #f
```

"<char>"

A singleton char set written as a string of length one rather than a character. Equivalent to its interpretation as a literal string match, but included to clarify it can be composed in cset-sres.

<char-set>

A SRFI 14 character set, which matches any character in the set. Note that currently there is no portable written representation of SRFI 14 character sets, which means that this pattern is typically generated programmatically, such as with a quasiquoted expression.

```
(regexp-partition `(+ ,char-set:vowels) "vowels")
  => ("v" "o" "w" "e" "ls")
```

Rationale: Many useful character sets are likely to be available as SRFI 14 char-sets, so it is desirable to reuse them in regular expressions. Since many Unicode character sets are extremely large, converting back and forth between an internal and external representation can be expensive, so the option of direct embedding is necessary. When a readable external representation is needed, char-set->sre can be used.

(char-set <string>)

(<string>)

The set of chars as formed by SRFI 14 (string->char-set).

Note that char-sets contain code points, not grapheme clusters, so any combining characters in will be inserted separately from any preceding base characters by string->char-set.

```
(regexp-matches '(* ("aeiou")) "oui") => #<regexp-match>
(regexp-matches '(* ("aeiou")) "ouais") => #f
(regexp-matches '(* ("e\x0301")) "e\x0301") => #<regexp-match>
(regexp-matches '("e\x0301") "e\x0301") => #f
(regexp-matches '("e\x0301") "e") => #<regexp-match>
(regexp-matches '("e\x0301") "\x0301") => #<regexp-match>
(regexp-matches '("e\x0301") "\x00E9") => #f
```

(char-range <range-spec> ...)

(/ <range-spec> ...)

Ranged char set. The is a list of strings and characters. These are flattened and grouped into pairs of characters, and all ranges formed by the pairs are included in the char set.

```
(regexp-matches '(* (/ "AZ09")) "R2D2") => #<regexp-match>
(regexp-matches '(* (/ "AZ09")) "C-3PO") => #f
```

(or <cset-sre> ...)

(|\|| <cset-sre> ...)

Char set union. The single vertical bar form is provided for consistency and compatibility with SCSH, although it needs to be escaped in R7RS.

(complement <cset-sre> ...)

(~ <cset-sre> ...)

Char set complement (i.e. [^...] in PCRE notation). (~ x) is equivalent to (- any x), thus in an ASCII context the complement is always ASCII.

(difference <cset-sre> ...)

(- <cset-sre> ...)

Char set difference.

```
(regexp-matches '(* (- (/ "az") ("aeiou"))) "xyzzy") => #<regexp-match>
(regexp-matches '(* (- (/ "az") ("aeiou"))) "vowels") => #f
```

(and <cset-sre> ...)

(& <cset-sre> ...)

Char set intersection.

```
(regexp-matches '(* (& (/ "az") (~ ("aeiou")))) "xyzzy") => #<regexp-match>
(regexp-matches '(* (& (/ "az") (~ ("aeiou")))) "vowels") => #f
```

any

Match any character. Equivalent to ascii in an ASCII context.

nonl

Match any character other than #or #
.

ascii

Match any ASCII character [0..127].

lower-case

lower

Matches any character for which char-lower-case? returns true. In a Unicode context this corresponds to the Lowercase (Ll + Other_Lowercase) property. In an ASCII context corresponds to (/ “az”).

upper-case

upper

Matches any character for which char-upper-case? returns true. In a Unicode context this corresponds to the Uppercase (Lu + Other_Uppercase) property. In an ASCII context corresponds to (/ “AZ”).

title-case

title

Matches any character with the Unicode Titlecase (Lt) property. This property only exists for the sake of ligature characters, of which only 31 exist at time of writing. In an ASCII context this is empty.

alphabetic

alpha

Matches any character for which char-alphabetic? returns true. In a Unicode context this corresponds to the Alphabetic (L + Nl + Other_Alphabetic) property. In an ASCII context corresponds to (w/nocase (/ “az”)).

numeric

num

Matches any character for which char-numeric? returns true. For In a Unicode context this corresponds to the Numeric_Digit (Nd) property. In an ASCII context corresponds to (/ “09”).

alphanumeric

alphanum

alnum

Matches any character which is either a letter or number. Equivalent to:

```
(or alphabetic numeric)
```

punctuation

punct

Matches any punctuation character. In a Unicode context this corresponds to the Punctuation property (P). In an ASCII context this corresponds to "!"#%&'()*,-./:;?@[]_{}".

symbol

Matches any symbol character. In a Unicode context this corresponds to the Symbol property (Sm, Sc, Sk, or So). In an ASCII this corresponds to "$+<=>^`|~".

graphic

graph

Matches any graphic character. Equivalent to:

```
(or alphanumeric punctuation symbol)
```

whitespace

white

space

Matches any whitespace character. In a Unicode context this corresponds to the Separator property (Zs, Zl or Zp). In an ASCII context this corresponds to space, tab, line feed, form feed, and carriage return.

printing

print

Matches any printing character. Equivalent to:

```
(or graphic whitespace)
```

control

cntrl

Matches any control or other character. In a Unicode context this corresponds to the Other property (Cc, Cf, Co, Cs or Cn). In an ASCII context this corresponds to:

```
`(/ ,(integer->char 0) ,(integer-char 31))
```

hex-digit

xdigit

Matches any valid digit in hexadecimal notation. Always ASCII-only. Equivalent to:

```
(w/ascii (w/nocase (or numeric "abcdef")))
```

bos

eos

Matches at the beginning/end of string without consuming any characters (a zero-width assertion). If the search was initiated with start/end parameters, these are considered the end points, rather than the full string.

bol

eol

Matches at the beginning/end of a line without consuming any characters (a zero-width assertion). A line is a possibly empty sequence of characters followed by an end of line sequence as understood by the R7RS read-line procedure, specifically any of a linefeed character, carriage return character, or a carriage return followed by a linefeed character. The string is assumed to contain end of line sequences before the start and after the end of the string, even if the search was made on a substring and the actual surrounding characters differ.

bow

eow

Matches at the beginning/end of a word without consuming any characters (a zero-width assertion). A word is a contiguous sequence of characters that are either alphanumeric or the underscore character, i.e. (or alphanumeric "_"), with the definition of alphanumeric depending on the Unicode or ASCII context. The string is assumed to contain non-word characters immediately before the

start and after the end, even if the search was made on a substring and word constituent characters appear immediately before the beginning or after the end.

```
(regexp-search '(: bow "foo") "foo") => #<regexp-match>
(regexp-search '(: bow "foo") "<foo>>") => #<regexp-match>
(regexp-search '(: bow "foo") "snafoo") => #f
(regexp-search '(: "foo" eow) "foo") => #<regexp-match>
(regexp-search '(: "foo" eow) "foo!") => #<regexp-match>
(regexp-search '(: "foo" eow) "foobar") => #f
```

nwb

Matches a non-word-boundary (i.e. in PCRE). Equivalent to (neg-look-ahead (or bow eow)).

(word sre ...)

Anchors a sequence to word boundaries. Equivalent to:

```
(: bow sre ... eow)
```

(word+ cset-sre ...)

Matches a single word composed of characters in the intersection of the given cset-sre and the word constituent characters. Equivalent to:

```
(word (+ (and (or alphanumeric "_") (or cset-sre ...))))
```

word

A shorthand for (word+ any).

bog

eog

Matches at the beginning/end of a single extended grapheme cluster without consuming any characters (a zero-width assertion). Grapheme cluster boundaries are defined in Unicode TR29. The string is assumed to contain non-combining codepoints immediately before the start and after the end. These always succeed in an ASCII context.

grapheme

Matches a single grapheme cluster (i.e. in PCRE). This is what the end-user typically thinks of as a single character, comprised of a base non-combining codepoint followed by zero or more combining marks. In an ASCII context this is equivalent to any.

Assuming char-set:mark contains all characters with the Extend or SpacingMark properties defined in TR29, and char-set:control, char-set:regional-indicator and char-set:hangul-* are defined similarly, then the following SRE can be used with regexp-extract to define grapheme:

```
`(or (: (* ,char-set:hangul-l) (+ ,char-set:hangul-v)
        (* ,char-set:hangul-t))
     (: (* ,char-set:hangul-l) ,char-set:hangul-v
        (* ,char-set:hangul-v) (* ,char-set:hangul-t))
     (: (* ,char-set:hangul-l) ,char-set:hangul-lvt
        (* ,char-set:hangul-t))
     (+ ,char-set:hangul-l)
     (+ ,char-set:hangul-t)
     (+ ,char-set:regional-indicator)
     (: "\r\n")
     (: (~ control ("\r\n"))
        (* ,char-set:mark))
     control)
```

(non-greedy-optional sre ...)

(?? sre ...)

Non-greedy pattern, matches 0 or 1 times, preferring the shorter match.

(non-greedy-zero-or-more< sre ...)

(*? sre ...)

Non-greedy Kleene star, matches 0 or more times, preferring the shorter match.

(non-greedy-repeated m n sre ...)

(**? m n sre ...)

Non-greedy Kleene star, matches m to n times, preferring the shorter match.

(look-ahead sre ...)

Zero-width look-ahead assertion. Asserts the sequence matches from the current position, without advancing the position.

```
(regexp-matches '(: "regular" (look-ahead " expression") " expression") "regular expression"
(regexp-matches '(: "regular" (look-ahead " ") "expression") "regular expression")=> #f
```

`(look-behind sre ...)`

Zero-width look-behind assertion. Asserts the sequence matches behind the current position, without advancing the position. It is an error if the sequence does not have a fixed length.

`(neg-look-ahead sre ...)`

Zero-width negative look-ahead assertion.

`(neg-look-behind sre ...)`

Zero-width negative look-behind assertion. # `(scheme mapping hash)`

`(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.

`(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.

(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.

(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.

`(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-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.

(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.

(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.

(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. # `(scheme ephemeron)`

This library is based on SRFI-124, that is itself based on the MIT Scheme Reference Manual.

An ephemeron is an object with two components called its key and its datum. It differs from an ordinary pair as follows: if the garbage collector (GC) can prove that there are no references to the key except from the ephemeron itself and possibly from the datum, then it is free to break the ephemeron, dropping its reference to both key and datum. In other words, an ephemeron can be broken when nobody else cares about its key. Ephemerons can be used to construct weak vectors or lists and (possibly in combination with finalizers) weak hash tables.

(ephemeron? obj)

Returns #t if object is an ephemeron; otherwise returns #f.

(make-ephemeron key datum)

Returns a newly allocated ephemeron, with components key and datum. Note that if key and datum are the same in the sense of eq?, the ephemeron is effectively a weak reference to the object.

(ephemeron-broken? ephemeron)

Returns #t if ephemeron has been broken; otherwise returns #f.

This procedure must be used with care. If it returns #f, that guarantees only that prior evaluations of ephemeron-key or ephemeron-datum yielded the key or datum that was stored in ephemeron. However, it makes no guarantees about subsequent calls to ephemeron-key or ephemeron-datum, because the GC may run and break the ephemeron immediately after ephemeron-broken? returns. Thus, the correct idiom to fetch an ephemeron's key and datum and use them if the ephemeron is not broken is:

```
(let ((key (ephemeron-key ephemeron))
      (datum (ephemeron-datum ephemeron)))
  (if (ephemeron-broken? ephemeron)
      ... broken case ...
      ... code using key and datum ...))
```

(ephemeron-key ephemeron)

(ephemeron-value ephemeron)

These return the key or datum component, respectively, of ephemeron. If ephemeron has been broken, these operations return #f, but they can also return #f if that is what was stored as the key or datum.

(reference-barrier key)

This procedure is optional.

This procedure ensures that the garbage collector does not break an ephemeron containing an unreferenced key before a certain point in a program. The program can invoke a reference barrier on the key by calling this procedure, which guarantees that even if the program does not use the key, it will be considered strongly reachable until after reference-barrier returns.