Revised^7 Report on the Algorithmic Language Scheme

Large Language, Red Edition

WG2 membership (Editors)

With major contributions from:

SRFI authors

Dedicated to the memory of ...?

DRAFT May 28, 2017

Authors

TODO Rework this whole thing. I'm tentatively thinking that the title page should list the editors, followed by "Major Contributions by", and the SRFI authors, in alphabetical order. Acknowledgements should appear in alphabetical order, in the document, with each person listed once, no matter how many SRFIs they are credited for. Not totally sure what to do about the copyright notices, since many of the original notices say "All Rights Reserved", and the standard SRFI notice contradicts that, and only talks about "the Software". Suggest writing to the SRFI authors and asking them if it's permissible to put the derived documentation under the R7RS copyright (R7RS, p3), which is in my non-legal opinion equivalent to public domain.

- SRFI 1: List library. Olin Shivers. `(scheme list)`

  Copyright (C) Olin Shivers (1998, 1999). All Rights Reserved.

  The design of this library benefited greatly from the feedback provided during the SRFI discussion phase. Among those contributing thoughtful commentary and suggestions, both on the mailing list and by private discussion, were Mike Ashley, Darius Bacon, Alan Bawden, Phil Bewig, Jim Blandy, Dan Bornstein, Per Bothner, Anthony Carrico, Doug Currie, Kent Dybvig, Sergei Egorov, Doug Evans, Marc Feeley, Matthias Felleisen, Will Fitzgerald, Matthew Flatt, Dan Friedman, Lars Thomas Hansen, Brian Harvey, Erik Hilsdale, Wolfgang Hukriede, Richard Kelsey, Donovan Kolbly, Shriram Krishnamurthi, Dave Mason, Jussi Piitulainen, David Pokorny, Duncan Smith, Mike Sperber, Maciej Stachowiak, Harvey J. Stein, John David Stone, and Joerg F. Wittenberger. I am grateful to them for their assistance.

  I am also grateful the authors, implementors and documentors of all the systems mentioned in the rationale. Aubrey Jaffer and Kent Pitman should be noted for their work in producing Web-accessible versions of the R5RS and Common Lisp spec, which was a tremendous aid.

  This is not to imply that these individuals necessarily endorse the final results, of course.

- SRFI 133: Vector Library. John Cowan (based on SRFI 43 by Taylor Campbell). `(scheme vector)`

  Copyright (C) Taylor Campbell (2003). All rights reserved. (And John Cowan?)

  These acknowledgements are copied from SRFI 43.

  Thanks to Olin Shivers for his wonderfully complete list and string packages; to all the members of the `#scheme` IRC channel on Freenode who nitpicked a great deal, but also helped quite a lot in general, and helped test the reference implementation in various Scheme systems; to Michael Burschik for his numerous comments; to Sergei Egorov for helping to narrow down the procedures; to Mike Sperber for putting up with an *extremely* overdue draft; to Felix Winkelmann for continually bugging me about finishing up the SRFI so that it would be only overdue and not withdrawn; and to everyone else who gave questions, comments, thoughts, or merely attention to the SRFI.

- SRFI 132: Sort Libraries. John Cowan (based on SRFI 32 by Olin Shivers)

  This document is copyright (C) Olin Shivers (1998, 1999). All Rights Reserved. (Copyright John Cowan as well?)

  Olin thanked the authors of the open source consulted when designing this library, particularly Richard O'Keefe, Donovan Kolbly and the MIT Scheme Team. John thanks Will Clinger for his detailed comments, and both Will Clinger and Alex Shinn for their implementation efforts.

- SRFI 113: Sets and Bags. John Cowan

  Copyright (C) John Cowan 2013. All Rights Reserved.

- SRFI 14: Character-set Library. Olin Shivers.

  Certain portions of this document – the specific, marked segments of text describing the R5RS procedures – were adapted with permission from the R5RS report.

  All other text is copyright (C) Olin Shivers (1998, 1999, 2000). All Rights Reserved.

  The design of this library benefited greatly from the feedback provided during the SRFI discussion phase. Among those contributing thoughtful commentary and suggestions, both on the mailing list and by private discussion, were Paolo Amoroso, Lars Arvestad, Alan Bawden, Jim Bender, Dan Bornstein, Per Bothner, Will Clinger, Brian Denheyer, Kent Dybvig, Sergei Egorov, Marc Feeley, Matthias Felleisen, Will Fitzgerald, Matthew Flatt, Arthur A. Gleckler, Ben Goetter, Sven Hartrumpf, Erik Hilsdale, Shiro Kawai, Richard Kelsey, Oleg Kiselyov, Bengt Kleberg, Donovan Kolbly, Bruce Korb, Shriram Krishnamurthi, Bruce Lewis, Tom Lord, Brad Lucier, Dave Mason, David Rush, Klaus Schilling, Jonathan Sobel, Mike Sperber, Mikael Staldal, Vladimir Tsyshevsky, Donald Welsh, and Mike Wilson. I am grateful to them for their assistance.

I am also grateful the authors, implementors and documentors of all the systems mentioned in the introduction. Aubrey Jaffer should be noted for his work in producing Web-accessible versions of the R5RS spec, which was a tremendous aid.

This is not to imply that these individuals necessarily endorse the final results, of course.

During this document's long development period, great patience was exhibited by Mike Sperber, who is the editor for the SRFI, and by Hillary Sullivan, who is not.

- SRFI 125: Intermediate hash tables. John Cowan, Will Clinger.

  Copyright (C) John Cowan (2015).

  Some of the language of this SRFI is copied from SRFI 69 with thanks to its author, Panu Kalliokoski. However, he is not responsible for what I have done with it. Thanks to Will Clinger for providing the sample implementation, and to Taylan Ulrich Bayırlı/Kammer for his spirited review.

  I also acknowledge the members of the SRFI 125, 126, and 128 mailing lists, especially Takashi Kato, Alex Shinn, Shiro Kawai, and Per Bothner.

- SRFI 116: Immutable List Library. John Cowan.

  Copyright (C) John Cowan 2014. All Rights Reserved.

  Without the work of Olin Shivers on SRFI 1, this SRFI would not exist. Everyone acknowledged there is transitively acknowledged here. This is not to imply that either Olin or anyone else necessarily endorses the final results, of course.

- SRFI 101: Purely Functional Random-Access Pairs and Lists. David Van Horn.

  Copyright (C) David Van Horn 2009. All Rights Reserved.

  I am grateful to the members of the Northeastern University Programming Research Laboratory and PLT (and their intersection) for discussions during the pre-draft development of this SRFI and the library that preceded it. We are all indebted to Okasaki for his elegant solution to this problem. Much of the specification is adapted from text intended to belong to the Scheme community; I thank the editors and authors of the RnRS series collectively for their efforts. I am grateful to Donovan Kolbly for serving as SRFI editor and to Taylor R Campbell, Robert Bruce Findler, Aubrey Jaffer, Shiro Kawai, and Alexey Radul for discussion during the draft period. I thank William D Clinger, Robert Bruce Findler, and Abdulaziz Ghuloum for help writing the implementations of `quote` specific to Larceny, Ikarus, and PLT, respectively. Support was provided by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellow Project.

- SRFI 134: Immutable Deques. Kevin Wortman, John Cowan.

  Copyright (C) John Cowan, Kevin Wortman (2015). All Rights Reserved.

- SRFI 135: Immutable Texts. William D Clinger.

  Copyright (C) William D Clinger (2016). All Rights Reserved.

  For three decades, I have been hoping the Scheme standards would either make strings immutable or add a new data type of immutable texts; with Unicode, that hope became more urgent. During that time, I have discussed this with far more people than I can now remember. Most of those I do remember are among those acknowledged below by John Cowan or Olin Shivers, but I am pleased to add Lars T Hansen, Chris Hanson, Felix Klock, and Jonathan Rees to the list of those whose ideas (and counter-arguments!) have contributed to this SRFI.

  John Cowan, the author of SRFI 130, deserves special thanks for blessing my desire to use SRFI 130 as the starting point for this SRFI, for designing the spans API whose implementations tested the key ideas of this SRFI's sample implementations, for chairing Working Group 2, and for a lot more I won't mention here.

  To acknowledge all those who contributed to SRFI 130 and to its predecessor SRFI 13, written by Olin Shivers, I hereby reproduce John Cowan's acknowledgements from SRFI 130:

Thanks to the members of the SRFI 130 mailing list who made this SRFI what it now is, including Per Bothner, Arthur Gleckler, Shiro Kawai, Jim Rees, and especially Alex Shinn, whose idea it was to make cursors and indexes disjoint, and who provided the foof implementation. The following acknowledgements by Olin Shivers are taken from SRFI 13:

> The design of this library benefited greatly from the feedback provided during the SRFI discussion phase. Among those contributing thoughtful commentary and suggestions, both on the mailing list and by private discussion, were Paolo Amoroso, Lars Arvestad, Alan Bawden, Jim Bender, Dan Bornstein, Per Bothner, Will Clinger, Brian Denheyer, Mikael Djurfeldt, Kent Dybvig, Sergei Egorov, Marc Feeley, Matthias Felleisen, Will Fitzgerald, Matthew Flatt, Arthur A. Gleckler, Ben Goetter, Sven Hartrumpf, Erik Hilsdale, Richard Kelsey, Oleg Kiselyov, Bengt Kleberg, Donovan Kolbly, Bruce Korb, Shriram Krishnamurthi, Bruce Lewis, Tom Lord, Brad Lucier, Dave Mason, David Rush, Klaus Schilling, Jonathan Sobel, Mike Sperber, Mikael Staldal, Vladimir Tsyshevsky, Donald Welsh, and Mike Wilson. I am grateful to them for their assistance.
> I am also grateful to the authors, implementors and documentors of all the systems mentioned in the introduction. Aubrey Jaffer and Kent Pitman should be noted for their work in producing Web-accessible versions of the R5RS and Common Lisp spec, which was a tremendous aid.
> This is not to imply that these individuals necessarily endorse the final results, of course.
> During this document's long development period, great patience was exhibited by Mike Sperber, who is the editor for the SRFI, and by Hillary Sullivan, who is not.

As Olin said, we should not assume any of those individuals endorse this SRFI.

- SRFI 121: Generators. Shiro Kawai, John Cowan, Thomas Gilray.

  Copyright (C) Shiro Kawai, John Cowan, Thomas Gilray (2015). All Rights Reserved.

  These procedures are drawn from the Gauche core and the Gauche module `gauche.generator`, with some renaming to make them more systematic, and with a few additions from the Python library `itertools`. Consequently, Shiro Kawai, the author of Gauche and its specifications, is listed as first author of this SRFI. John Cowan served as editor and shepherd. Thomas Gilray provided the sample implementation and a valuable critique of the SRFI. Special acknowledgements to Kragen Javier Sitaker for his extensive review.

- SRFI 127: Lazy Sequences. John Cowan.

  Copyright (C) John Cowan (2015). All Rights Reserved.

  Without the work of Olin Shivers on SRFI 1, this SRFI would not exist. Everyone acknowledged there is transitively acknowledged here. This is not to imply that either Olin or anyone else necessarily endorses the final results, of course.

  Special thanks to Shiro Kawai, whose Gauche implementation of lazy sequences inspired this one, and to Kragen Javier Sitaker, who did a thorough review.

- SRFI 41: Streams. Philip L. Bewig.

  Copyright (C) Philip L. Bewig (2007). All Rights Reserved.

  Jos Koot sharpened my thinking during many e-mail discussions, suggested several discussion points in the text, and contributed the final version of stream-match. Michael Sperber and Abdulaziz Ghuloum gave advice on R6RS.

- SRFI 111: Boxes. John Cowan.

  Copyright (C) John Cowan 2013. All Rights Reserved.

- SRFI 117: Queues based on lists. John Cowan.

  Copyright © John Cowan, 2014. All Rights Reserved.

- SRFI 124: Ephemerons.John Cowan.

  Copyright (C) John Cowan (2015).

- SRFI 128: Comparators (reduced). John Cowan.

  Copyright (C) John Cowan (2015). All Rights Reserved.

  This SRFI is a simplified and enhanced rewrite of SRFI 114, and shares some of its design rationale and all of its acknowledgements. The largest change is the replacement of the comparison procedure with the ordering procedure. This allowed most of the special-purpose comparators to be removed. In addition, many of the more specialized procedures, as well as all but one of the syntax forms, have been removed as unnecessary.

  Special thanks to Taylan Ulrich Bayırlı/Kammer, whose insistence that SRFI 114 was unacceptable inspired this redesign. Jörg Wittenberger added Chicken-specific type declarations, which I have moved to `comparators.scm`, as it is a Chicken-specific library. He also provided Chicken-specific metadata and setup commands. Comments from Shiro Kawai, Alex Shinn, and Kevin Wortman guided me to the current design for bounds and salt.

Others: WG2 members.

Copyright notice

TODO This is really a report, rather than software. I think the statement below should be replaced by the (non-copyright) from the R^{7}RS report: “We intend this report to belong to the entire Scheme com- munity, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementers of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.”. The individual SRFI comments can be included as a list, along with a statement that says that each author has explicitly granted permission for free use of their work.

TODO Restore 2-column layout, once LaTeX artifacts have been fixed.

SUMMARY

[This needs to be rewritten.]

This is a frozen version of the Red Edition ballot.

By unanimous consent, the libraries of R7RS-small are all required in implementations of the Red Edition. This document specifies the additional libraries required in the Red Edition.

CONTENTS

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. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

The Revised^7 Report on The Algorithmic Language Scheme (R^{7}RS) describes an elegant and powerful language that fulfills the promise of the previous paragraph. In some cases, e.g., when using Scheme as an extension language, that is all the programmer needs. Often, though, modern programmers expect a set of libraries that will provide the facilities they need to get the job done. Many Scheme implementations have provided such libraries, but often the libraries are implementation-specific. It would be desirable to standardize as many of them as possible.

Accordingly, the Scheme community embarked upon the collection of a group of Scheme Requests For Implementation, or SRFIs. These specify particular libraries that can be provided in a compatible way across implementations.

More recently, it was decided that R^{7}RS would be a core language, dubbed "R^{7}RS-Small", and that a Large language would be created by identifying particular SRFIs, and adapting them to form a consistent library collection.

This report, the Red Edition, is the first stage of defining R^{7}RS-Large, concentrating primarily upon data structure support. Subsequent reports, also named for colors, will provide additional facilities.

TODO Check all descriptions to make sure that argument references are in appropriate font.

1. Basic libraries

List library

[**Red Edition item 1**] The small language has a basic set of list-processing utilities. The large language extends this set substantially.

The names described in this section comprise the `(scheme list)` library.

TODO Some of these procedures duplicate those of the small language. Do we want that?

TODO Define "linear update" and "pure functional" procedures. I assume that other SRFIs use the same terms, so maybe in a "Definitions" section in the introduction?

Procedure Index

TODO I think this whole section can be deleted. There are a few comments here about how a procedure in this library extends one in Scheme-small. Those should probably be folded into the descriptions of the individual procedures.

Here is a short list of the procedures provided.

Constructors `cons list`
```
    xcons cons* make-list list-tabulate
    list-copy circular-list iota
```

Predicates `pair? null?`
```
    proper-list? circular-list? dotted-list?
    not-pair? null-list?
    list=
```

Selectors `car cdr ... cddadr cddddr list-ref`
```
    first second third fourth fifth
    sixth seventh eighth ninth tenth
    car+cdr
    take       drop
    take-right drop-right
    take!      drop-right!
    split-at   split-at!
    last last-pair
```

Miscellaneous: length, append, concatenate, reverse, zip & count `length length+`

```
append  concatenate  reverse
append! concatenate! reverse!
append-reverse append-reverse!
zip unzip1 unzip2 unzip3 unzip4 unzip5
count
```

Fold, unfold & map `map for-each`

```
fold       unfold       pair-fold
fold-right unfold-right pair-fold-right
reduce  reduce-right
append-map append-map!
map! pair-for-each filter-map map-in-order
```

Filtering & partitioning `filter  partition  remove`

```
filter! partition! remove!
```

Searching `member memq memv`

```
find find-tail
any every
list-index
take-while drop-while take-while!
span break span! break!
```

Deleting `delete  delete-duplicates`

```
delete! delete-duplicates!
```

Association lists `assoc assq assv`

```
alist-cons alist-copy
alist-delete alist-delete!
```

Set operations on lists `lset<= lset= lset-adjoin`

```
lset-union              lset-union!
lset-intersection       lset-intersection!
lset-difference         lset-difference!
lset-xor                lset-xor!
lset-diff+intersection lset-diff+intersection!
```

Primitive side-effects `set-car! set-cdr!`

Four R4RS/R5RS list-processing procedures are extended by this library in backwards-compatible ways:

`map for-each`	Extended to take lists of unequal length
`member assoc`	Extended to take an optional comparison procedure.

The following R4RS/R5RS list- and pair-processing procedures are also part of list-lib's exports, as defined by the R5RS:

```
cons pair? null?
car cdr ... cdddar cddddr
set-car! set-cdr!
list append reverse
length list-ref
memq memv assq assv
```

The remaining two R4RS/R5RS list-processing procedures are *not* part of this library:

`list-tail`	renamed `drop`
`list?`	see `proper-list?`, `circular-list?` and `dotted-list?`

The linear-update procedures in this library are

`take! drop-right! split-at! append! concatenate! reverse! append-reverse! append-map! map! filter! partition! remove! take-while! span! break! delete! alist-delete! delete-duplicates! lset-adjoin! lset-union! lset-inters lset-difference! lset-xor! lset-diff+intersection!`

Improper Lists

Scheme does not properly have a list type, just as C does not have a string type. Rather, Scheme has a binary-tuple type, from which one can build binary trees. There is an *interpretation* of Scheme values that allows one to treat these trees as lists. Further complications ensue from the fact that Scheme allows side-effects to these tuples, raising the possibility of lists of unbounded length, and trees of unbounded depth (that is, circular data structures).

However, there is a simple view of the world of Scheme values that considers every value to be a list of some sort. that is, every value is either

- a "proper list" – a finite, nil-terminated list, such as:
  ```
  (a b c)
  ()
  (32)
  ```
- a "dotted list" – a finite, non-nil terminated list, such as:
  ```
  (a b c . d)
  (x . y)
  42
  george
  ```
- or a "circular list" – an infinite, unterminated list.

Note that the zero-length dotted lists are simply all the non-null, non-pair values.

This view is captured by the predicates `proper-list?`, `dotted-list?`, and `circular-list?`. List-lib users should note that dotted lists are not commonly used, and are considered by many Scheme programmers to be an ugly artifact of Scheme's lack of a true list type. However, dotted lists do play a noticeable role in the *syntax* of Scheme, in the "rest" parameters used by n-ary lambdas: `(lambda (x y . rest) ...)`.

Dotted lists are *not* fully supported by list-lib. Most procedures are defined only on proper lists – that is, finite, nil-terminated lists. The procedures that will also handle circular or dotted lists are specifically marked. While this design decision restricts the domain of possible arguments one can pass to these procedures, it has the benefit of allowing the procedures to catch the error cases where programmers inadvertently pass scalar values to a list procedure by accident, *e.g.*, by switching the arguments to a procedure call.

Errors

TODO Rewrite this section so it matches R^{7}RS. Ideally, nothing additional regarding errors should need to appear in this Report, because the error language should match R^{7}RS.

Note that statements of the form "it is an error" merely mean "don't do that." They are not a guarantee that a conforming implementation will "catch" such improper use by, for example, raising some kind of exception. Regrettably, R5RS Scheme requires no firmer guarantee even for basic operators such as `car` and `cdr`, so there's little point in requiring these procedures to do more. Here is the relevant section of the R5RS:

> When speaking of an error situation, this report uses the phrase "an error is signalled" to indicate that implementations must detect and report the error. If such wording does not appear in the discussion of an error, then implementations are not required to detect or report the error, though they are encouraged to do so. An error situation that implementations are not required to detect is usually referred to simply as "an error."

For example, it is an error for a procedure to be passed an argument that the procedure is not explicitly specified to handle, even though such domain errors are seldom mentioned in this report. Implementations may extend a procedure's domain of definition to include such arguments.

Notation

TODO Move this to the beginning of the Report, as part of a standardized "Notation" section, and reformat. Check that these notations are consistent with those in R^7RS.

list	A proper (finite, nil-terminated) list
clist	A proper or circular list
flist	A finite (proper or dotted) list
pair	A pair
x, y, d, a	Any value
object, value	Any value
n, i	A natural number (an integer >= 0)
proc	A procedure
pred	A procedure whose return value is treated as a boolean
=	A boolean procedure taking two arguments

It is an error to pass a circular or dotted list to a procedure not defined to accept such an argument.

Constructors

`(cons` *a d*`)` → *pair* list library procedure

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.

```
(cons 'a '())        => (a)
(cons '(a) '(b c d)) => ((a) b c d)
(cons "a" '(b c))    => ("a" b c)
(cons 'a 3)          => (a . 3)
(cons '(a b) 'c)     => ((a b) . c)
```

`(list` *object ...*`)` → *list* list library procedure

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c) =>  (a 7 c)
(list)               =>  ()
```

`(xcons` *d a*`)` → *pair* list library procedure

```
(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*` *elt_1 elt_2 ...*`)` → *object* list library procedure

Like `list`, but the last argument provides the tail of the constructed list, returning

```
 (cons elt1 (cons elt2 (cons ... eltn)))
```

This function is called `list*` in Common Lisp and about half of the Schemes that provide it, and `cons*` in the other half.

```
(cons* 1 2 3 4) => (1 2 3 . 4)
(cons* 1) => 1
```

`(make-list` *n fill*`)` → *list* list library procedure
`(make-list` *n*`)` → *list* list library procedure

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*`)` → *list* list library procedure

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*`)` → *flist* list library procedure

Copies the spine of the argument. TODO More detail.

`(circular-list` *elt*$_1$ *elt*$_2$ *...*`)` → *list* list library procedure

Constructs a circular list of the elements.

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

`(iota` *count start step*`)` → *list* list library procedure
`(iota` *count*`)` → *list* list library procedure

Returns a list 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.

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

Predicates

Note: the predicates `proper-list?`, `circular-list?`, and `dotted-list?` partition the entire universe of Scheme values.

`(proper-list?` *x*`)` → *boolean* list library procedureReturns true iff x is a proper list – a finite, nil-terminated list.

TODO I'm not sure where this stuff should go. Maybe at the beginning of the section?

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

```
<proper-list> ::= ()                            (Empty proper list)
              |   (cons <x> <proper-list>)      (Proper-list pair)
```

Note that this definition rules out circular lists. This function is required to detect this case and return false.

Nil-terminated lists are called "proper" lists by R5RS and Common Lisp. The opposite of proper is improper.

R5RS binds this function to the variable `list?`.

```
(not (proper-list? x)) = (or (dotted-list? x) (circular-list? x))
```

TODO This needs to be rewritten; definitions should be moved to a “Definitions” section at the front.

`(circular-list? x)` → *boolean* list library procedure

True if x is a circular list. A circular list is a value such that for every n >= 0, $\mathrm{cdr}^n(x)$ is a pair.

Terminology: The opposite of circular is finite.

```
(not (circular-list? x)) = (or (proper-list? x) (dotted-list? x))
```

`(dotted-list? x)` → *boolean* list library procedure

True if x is a finite, non-nil-terminated list. That is, there exists an n >= 0 such that $\mathrm{cdr}^n(x)$ 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.

```
(not (dotted-list? x)) = (or (proper-list? x) (circular-list? x))
```

`(pair? object)` → *boolean* list library procedure

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

```
(pair? '(a . b)) =>  #t
(pair? '(a b c)) =>  #t
(pair? '())      =>  #f
(pair? '#(a b))  =>  #f
(pair? 7)        =>  #f
(pair? 'a)       =>  #f
```

`(null? object)` → *boolean* list library procedure

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

`(null-list? list)` → *boolean* list library procedure

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)` → *boolean* list library procedure

```
(lambda (x) (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_1 elt_2 ...`)` → *boolean* list library procedure

Determines list equality, given an element-equality procedure. Proper list A equals proper list 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 list_i, then its second argument is from list_{i+1}, *i.e.* it is always called as `(elt= a b)` for a an element of list A, and b an element of list B.

In the n-ary case, every list_i is compared to list_{i+1} (as opposed, for example, to comparing list_1 to every list_i, for i>1). If there are no list arguments at all, `list=` simply returns true.

It is an error to apply `list=` to anything except proper lists. While implementations may choose to extend it to circular lists, note that it cannot reasonably be extended to dotted lists, as it provides no way to specify an equality procedure for comparing the list terminators.

Note that the dynamic order in which the elt= procedure is applied to pairs of elements is not specified. For example, if `list=` is applied to three lists, 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 lists which are `eq?` are always list=, as well; implementations may exploit this fact to "short-cut" the element-by-element comparisons.

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

Selectors

`(car` *pair*`)` → *boolean* list library procedure
`(cdr` *pair*`)` → *boolean* list library procedure

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.

```
(car '(a b c))     =>  a         (cdr '(a b c))     =>  (b c)
(car '((a) b c d)) =>  (a)       (cdr '((a) b c d)) =>  (b c d)
(car '(1 . 2))     =>  1         (cdr '(1 . 2))     =>  2
(car '())          =>  *error*   (cdr '())          =>  *error*
```

`(caar` *pair*`)` → *boolean* list library procedure
`(cadr` *pair*`)` → *boolean* list library procedure
:
`(cdddar` *pair*`)` → *boolean* list library procedure
`(cddddr` *pair*`)` → *boolean* list library procedure

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

`(define caddr (lambda (x) (car (cdr (cdr x))))).`

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

`(list-ref` *clist i*`)` → *value* list library procedure

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*`)` → *value* list library procedure
`(second` *pair*`)` → *value* list library procedure
`(third` *pair*`)` → *value* list library procedure
`(fourth` *pair*`)` → *value* list library procedure
`(fifth` *pair*`)` → *value* list library procedure
`(sixth` *pair*`)` → *value* list library procedure
`(seventh` *pair*`)` → *value* list library procedure
`(eighth` *pair*`)` → *value* list library procedure
`(nineth` *pair*`)` → *value* list library procedure
`(tenth` *pair*`)` → *value* list library procedure

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

```
(third '(a b c d e)) => c
```

`(car+cdr` *pair*`)` → *x y* list library procedure

Returns the car and cdr of pair as two values.

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

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

`(take` *x i*`)` → *list* list library procedure
`(drop` *x i*`)` → *object* list library procedure

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

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

x 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`:

```
(append (take x i) (drop x i)) = x
```

`drop` is exactly equivalent to performing i cdr operations on x; the returned value shares a common tail with x. 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 lis (length lis))`.

`(take-right` *flist i*`)` → *object* list library procedure
`(drop-right` *flist i*`)` → *list* list library procedure

`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`:

```
(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 lis 0)`.

`(take!` *x i*`)` → *list* list library procedure
`(drop-right!` *flist i*`)` → *list* list library procedure

`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*`)` → *list object* list library procedure
`(split-at!` *x i*`)` → *list object* list library procedure

`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)
    (d e f g h)
```

`(last` *pair*`)` → *object* list library procedure
`(last-pair` *pair*`)` → *pair* list library procedure

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

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

Miscellaneous

`(length` *list*`)` → *integer* list library procedure
`(length+` *list*`)` → *integer* or `#f` list library procedure

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` *list*$_1$ *...*`)` → *list* list library procedure
`(append!` *list*$_1$ *...*`)` → *list* list library procedure

`append` returns a list consisting of the elements of list$_1$ 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$_i$ 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*`)` → *value* list library procedure
`(concatenate!` *list-of-lists*`)` → *value* list library procedure

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`.

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 `(apply append ...)` idiom would fail when applied to long lists, but `concatenate` would continue to function properly.

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

`(reverse` *list*`)` → *list* list library procedure
`(reverse!` *list*`)` → *list* list library procedure

`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*`)` → *list* list library procedure
`(append-reverse!` *rev-head tail*`)` → *list* list library procedure

`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` *clist*$_1$ *clist*$_2$ *...*`)` → *list* list library procedure

```
(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))
```

At least one of the argument lists must be finite:

```
(zip '(3 1 4 1) (circular-list #f #t))
    => ((3 #f) (1 #t) (4 #f) (1 #t))
```

`(unzip1` *list*`)` → *list* list library procedure
`(unzip2` *list*`)` → *list list* list library procedure
`(unzip3` *list*`)` → *list list list* list library procedure
`(unzip4` *list*`)` → *list list list list* list library procedure
`(unzip5` *list*`)` → *list list list list list* list library procedure

`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` *clist*$_1$ *clist*$_2$ ...`)` → *integer* list library procedure

TODO The ellipsis wasn't present in the original. Is this a mistake?

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

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

At least one of the argument lists must be finite:

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

Fold, unfold, and map

`(fold` *kons knil clist*$_1$ *clist*$_2$ ...`)` → *value* list library procedure

The fundamental list iterator.

First, consider the single list-parameter case. If clist$_1$ = (e$_1$ e$_2$ ... e$_n$), then this procedure returns

```
(kons en ... (kons e2 (kons e1 knil)) ... )
```

That is, it obeys the (tail) recursion

```
(fold kons knil lis) = (fold kons (kons (car lis) knil) (cdr lis))
(fold kons knil '()) = knil
```

Examples:

```
(fold + 0 lis)          ; Add up the elements of LIS.

(fold cons '() lis)     ; Reverse LIS.

(fold cons tail rev-head)   ; See APPEND-REVERSE.

;; How many symbols in LIS?
```

```
(fold (lambda (x count) (if (symbol? x) (+ count 1) count))
      0
      lis)

;; Length of the longest string in LIS:
(fold (lambda (s max-len) (max max-len (string-length s)))
      0
      lis)
```

If n list arguments are provided, then the kons function must take n+1 parameters: one element from each list, and the "seed" or fold state, which is initially knil. The fold operation terminates when the shortest list runs out of values:

```
(fold cons* '() '(a b c) '(1 2 3 4 5)) => (c 3 b 2 a 1)
```

At least one of the list arguments must be finite.

`(fold-right` *kons knil* $clist_1$ $clist_2$ *...*`)` → *value* list library procedure

The fundamental list recursion operator.

First, consider the single list-parameter case. If $clist_1$ = `(e1 e2 ... en)`, then this procedure returns

```
 (kons e1 (kons e2 ... (kons en knil)))
```

That is, it obeys the recursion

```
(fold-right kons knil lis) = (kons (car lis) (fold-right kons knil (cdr lis)))
(fold-right kons knil '()) = knil
```

Examples:

```
(fold-right cons '() lis)       ; Copy LIS.

;; Filter the even numbers out of LIS.
(fold-right (lambda (x l) (if (even? x) (cons x l) l)) '() lis))
```

If n list arguments are provided, then the kons function must take n+1 parameters: one element from each list, and the "seed" or fold state, which is initially knil. The fold operation terminates when the shortest list runs out of values:

```
(fold-right cons* '() '(a b c) '(1 2 3 4 5)) => (a 1 b 2 c 3)
```

At least one of the list arguments must be finite.

`(pair-fold` *kons knil* $clist_1$ $clist_2$ *...*`)` → *value* list library procedure

Analogous to `fold`, but kons is applied to successive sublists of the lists, rather than successive elements – that is, kons is applied to the pairs making up the lists, giving this (tail) recursion:

```
(pair-fold kons knil lis) = (let ((tail (cdr lis)))
                              (pair-fold kons (kons lis knil) tail))
(pair-fold kons knil '()) = knil
```

For finite lists, the kons function may reliably apply `set-cdr!` to the pairs it is given without altering the sequence of execution.

Example:

```
;;; Destructively reverse a list.
(pair-fold (lambda (pair tail) (set-cdr! pair tail) pair) '() lis))
```

At least one of the list arguments must be finite.

`(pair-fold-right` *kons knil* $clist_1$ $clist_2$ *...*`)` → *value* list library procedure

Holds the same relationship with `fold-right` that `pair-fold` holds with `fold`. Obeys the recursion

```
(pair-fold-right kons knil lis) =
    (kons lis (pair-fold-right kons knil (cdr lis)))
(pair-fold-right kons knil '()) = knil
```

Example:

```
(pair-fold-right cons '() '(a b c)) => ((a b c) (b c) (c))
```

At least one of the list arguments must be finite.

(`reduce` *f ridentity list*) → *value* list library procedure

`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
```

`reduce` has the following definition:

If list = (), return ridentity;
Otherwise, return `(fold f (car list) (cdr list))`.

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

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`).

Note: MIT Scheme and Haskell flip F's arg order for their `reduce` and `fold` functions.

```
;; Take the max of a list of non-negative integers.
(reduce max 0 nums) ; i.e., (apply max 0 nums)
```

(`reduce-right` *f ridentity list*) → *value* list library procedure

`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*) → *list* list library procedure
(`unfold` *p f g seed*) → *list* list library procedure

`unfold` is best described by its basic recursion:

```
(unfold p f g seed) =
    (if (p seed) (tail-gen seed)
        (cons (f seed)
              (unfold p f g (g seed))))
```

p Determines when to stop unfolding.

f Maps each seed value to the corresponding list element.
g Maps each seed value to next seed value.
seed The "state" value for the unfold.
tail-gen Creates the tail of the list; defaults to `(lambda (x) '())`

In other words, we use g to generate a sequence of seed values

seed, g(seed), g^2(seed), g^3(seed), ...

These seed values are mapped to list elements by f, producing the elements of the result list in a left-to-right order. P says when to stop.

`unfold` is the fundamental recursive list constructor, just as `fold-right` is the fundamental recursive list consumer. While `unfold` may seem a bit abstract to novice functional programmers, it can be used in a number of ways:

```
;; List of squares: 1^2 ... 10^2
(unfold (lambda (x) (> x 10))
        (lambda (x) (* x x))
    (lambda (x) (+ x 1))
    1)

(unfold null-list? car cdr lis) ; Copy a proper list.

;; Read current input port into a list of values.
(unfold eof-object? values (lambda (x) (read)) (read))

;; Copy a possibly non-proper list:
(unfold not-pair? car cdr lis
              values)

;; Append HEAD onto TAIL:
(unfold null-list? car cdr head
              (lambda (x) tail))
```

Interested functional programmers may enjoy noting that `fold-right` and `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

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

and

`(unfold knull? kar kdr (fold-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."

`(unfold-right` *p f g seed tail*`)` → *list* list library procedure
`(unfold-right` *p f g seed*`)` → *list* list library procedure

`unfold-right` constructs a list with the following loop:

```
(let lp ((seed seed) (lis tail))
  (if (p seed) lis
      (lp (g seed)
          (cons (f seed) lis))))
```

p Determines when to stop unfolding.
f Maps each seed value to the corresponding list element.

g Maps each seed value to next seed value.
seed The "state" value for the unfold.
tail list terminator; defaults to `'()`.

In other words, we use g to generate a sequence of seed values

seed, g(seed), g^2(seed), g^3(seed), ...

These seed values are mapped to list elements by f, producing the elements of the result list in a right-to-left order. P says when to stop.

`unfold-right` is the fundamental iterative list constructor, just as `fold` is the fundamental iterative list consumer. While `unfold-right` may seem a bit abstract to novice functional programmers, it can be used in a number of ways:

```
;; List of squares: 1^2 ... 10^2
(unfold-right zero?
              (lambda (x) (* x x))
              (lambda (x) (- x 1))
              10)

;; Reverse a proper list.
(unfold-right null-list? car cdr lis)

;; Read current input port into a list of values.
(unfold-right eof-object? values (lambda (x) (read)) (read))

;; (append-reverse rev-head tail)
(unfold-right null-list? car cdr rev-head tail)
```

Interested functional programmers may enjoy noting that `fold` and `unfold-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

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

and

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

~~This combinator presumably has some pretentious mathematical name; interested readers are invited to communicate it to the author.~~ TODO Do we want to say anything about this?

`(map` *proc clist*$_1$ *clist*$_2$ ...) → *list* list library procedure

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)
```

This procedure is extended from its R5RS specification to allow the arguments to be of unequal length; it terminates when the shortest list runs out.

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* $clist_1$ $clist_2$ ...`)` → *unspecified* list library procedure

[R5RS+] 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)
```

This procedure is extended from its R5RS specification to allow the arguments to be of unequal length; it terminates when the shortest list runs out.

At least one of the argument lists must be finite.

`(append-map` *f* $clist_1$ $clist_2$ ...`)` → *value* list library procedure
`(append-map!` *f* $clist_1$ $clist_2$ ...`)` → *value* list library procedure

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* $list_1$ $clist_2$ ...`)` → *list* list library procedure

Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of $list_1$ 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, $clist_2$, $clist_3$, ... must have at least as many elements as $list_1$.

`(map-in-order` *f* $clist_1$ $clist_2$ ...`)` → *list* list library procedure

A variant of the `map` procedure that guarantees to apply f across the elements of the $list_i$ 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* $clist_1$ $clist_2$ ...`)` → *unspecific* list library procedure

TODO I think the above should be “unspecified”, but in the original it says “unspecific”. Correct?

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 clist*$_1$ *clist*$_2$ *...*`)` → *list* list library procedure

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

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

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

At least one of the list arguments must be finite.

Filtering and partitioning

TODO introduction

`(filter` *pred list*`)` → *list* list library procedure

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*`)` → *list list* list library procedure

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*`)` → *list* list library procedure

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*`)` → *list* list library procedure
`(partition!` *pred list*`)` → *list list* list library procedure
`(remove!` *pred list*`)` → *list* list library procedure

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

Searching

TODO reorganize this introductory material, some of it belongs in the procedure entries.

The following procedures all search lists for a leftmost element satisfying some criteria. This means they do not always examine the entire list; thus, there is no efficient way for them to reliably detect and signal an error when passed a dotted or circular list. Here are the general rules describing how these procedures work when applied to different kinds of lists:

Proper lists: The standard, canonical behavior happens in this case.

Dotted lists: It is an error to pass these procedures a dotted list that does not contain an element satisfying the search criteria. That is, it is an error if the procedure has to search all the way to the end of the dotted list. However, this SRFI does *not* specify anything at all about the behavior of these procedures when passed a dotted list containing an element satisfying the search criteria. It may finish successfully, signal an error, or perform some third action. Different implementations may provide different functionality in this case; code which is compliant with this SRFI may not rely on any particular behavior. Future SRFI's may refine SRFI-1 to define specific behavior in this case. In brief, SRFI-1 compliant code may not pass a dotted list argument to these procedures.

Circular lists: It is an error to pass these procedures a circular list that does not contain an element satisfying the search criteria. Note that the procedure is not required to detect this case; it may simply diverge. It is, however, acceptable to search a circular list *if the search is successful* – that is, if the list contains an element satisfying the search criteria.

Here are some examples, using the `find` and `any` procedures as canonical representatives:

```
;; Proper list -- success
(find even? '(1 2 3))   => 2
(any  even? '(1 2 3))   => #t

;; proper list -- failure
(find even? '(1 7 3))   => #f
(any  even? '(1 7 3))   => #f

;; Failure is error on a dotted list.
(find even? '(1 3 . x)) => error
(any  even? '(1 3 . x)) => error

;; The dotted list contains an element satisfying the search.
;; This case is not specified -- it could be success, an error,
;; or some third possibility.
(find even? '(1 2 . x)) => error/undefined
(any  even? '(1 2 . x)) => error/undefined ; success, error or other.

;; circular list -- success
(find even? (circular-list 1 6 3)) => 6
(any  even? (circular-list 1 6 3)) => #t

;; circular list -- failure is error. Procedure may diverge.
(find even? (circular-list 1 3)) => error
(any  even? (circular-list 1 3)) => error
```

(`find` *pred clist*) → *value* list library procedure

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:

```
(cond ((find-tail pred lis) => (lambda (pair) ...)) ; Handle (CAR PAIR)
      (else ...)) ; Search failed.
```

`(find-tail` *pred clist*`)` → *pair or* `#f` list library procedure

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*`)` → *list* list library procedure
`(take-while!` *pred clist*`)` → *list* list library procedure

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*`)` → *list* list library procedure

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*`)` → *list clist* list library procedure
`(span!` *pred list*`)` → *list list* list library procedure
`(break` *pred clist*`)` → *list clist* list library procedure
`(break!` *pred list*`)` → *list list* list library procedure

`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* *clist*$_1$ *clist*$_2$ *...*) → *value* list library procedure

Applies the predicate across the lists, returning true TODO `#t`? if the predicate returns true on any application.

If there are n list arguments clist$_1$ … clist$_n$, 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 clist$_i$ parameters. If this application returns a true value, `any` immediately returns that value. Otherwise, it iterates, applying pred to the second elements of the clist$_i$ 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* *clist*$_1$ *clist*$_2$ *...*) → *value* list library procedure

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

If there are n list arguments clist$_1$ … clist$_n$, 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 clist$_i$ parameters. If this application returns false, `every` immediately returns false. Otherwise, it iterates, applying pred to the second elements of the clist$_i$ 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 clist$_i$ 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* *clist*$_1$ *clist*$_2$) → *integer or* `#f` list library procedure

Return the index of the leftmost element that satisfies pred.

If there are n list arguments clist$_1$ … clist$_n$, 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 clist$_i$ parameters. If this application returns true, `list-index` immediately returns zero. Otherwise, it iterates, applying pred to the second elements of the clist$_i$ 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 =*`)` → *list* list library procedure
`(member` *x list*`)` → *list* list library procedure
`(memq` *x list*`)` → *list* list library procedure
`(memv` *x list*`)` → *list* list library procedure

[R5RS+] 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)
```

`member` is extended from its R5RS definition to allow the client to pass in an optional equality procedure = used to compare keys.

The comparison procedure is used to compare the elements e_i 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.
```

Deletion

`(delete` *x list =*`)` → *list* list library procedure
`(delete` *x list*`)` → *list* list library procedure
`(delete!` *x list =*`)` → *list* list library procedure???
`(delete!` *x list*`)` → *list* list library procedure???

`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 e_i 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 =*`)` → *list* list library procedure
`(delete-duplicates` *list*`)` → *list* list library procedure
`(delete-duplicates!` *list =*`)` → *list* list library procedure
`(delete-duplicates!` *list*`)` → *list* list library procedure

`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(n^2) for n-element lists. Uniquifying long lists can be accomplished in O(n lg n) time by sorting the list to bring equal elements together, then using a linear-time algorithm to remove equal elements. Alternatively, one can use algorithms based on element-marking, with linear-time results.

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

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

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

Association lists

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

`(assoc` *key alist =*`)` → *pair or #f* list library procedure
`(assoc` *key alist*`)` → *pair or #f* list library procedure
`(assq` *key alist*`)` → *pair or #f* list library procedure
`(assv` *key alist*`)` → *pair or #f* list library procedure

[R5RS+] 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)
```

`assoc` is extended from its R5RS definition to allow the client to pass in an optional equality procedure = used to compare keys.

The comparison procedure is used to compare the elements e_i 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*`)` → *alist* list library procedure

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

Cons a new alist entry mapping key to datum onto alist.

`(alist-copy` *alist*`)` → *alist* list library procedure

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* list library procedure
`(alist-delete` *key alist*`)` → *alist* list library procedure
`(alist-delete!` *key alist =*`)` → *alist* list library procedure???
`(alist-delete!` *key alist*`)` → *alist* list library procedure???

`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 k_i of alist's entries to the key parameter in this way: `(= key ki)`. Thus, one can reliably remove all entries of alist whose key is greater than five with `(alist-delete 5 alist <)`

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

Set operations on lists

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

`(eq? x y)` => `(= x y)`.

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

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

`(lset<=` = *list*$_1$ *...*`)` → *boolean* list library procedure

Returns true iff every $list_i$ is a subset of $list_{i+1}$, 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= ` $=$ *list*$_1$ *list*$_2$ *...*`)` → *boolean* list library procedure

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

```
(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* *elt*$_1$ *...*`)` → *list* list library procedure

Adds the elt$_i$ 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 elt$_i$ is already a member of list. Its first argument is an element of list; its second is one of the elt$_i$.

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 ` $=$ *list*$_1$ *...*`)` → *list* list library procedure

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` = *list*$_1$ *list*$_2$ ...`)` → *list* list library procedure

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 list$_1$ – that is, `lset-intersection` essentially filters list$_1$, without disarranging element order. The result may share a common tail with list$_1$.

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 list$_1$ for membership in every other list before proceeding to consider the next element of list$_1$, or it may completely intersect list$_1$ and list$_2$ before proceeding to list$_3$, 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` = *list*$_1$ *list*$_2$ ...`)` → *list* list library procedure

Returns the difference of the lists, using = for the element-equality procedure – all the elements of list$_1$ that are not = to any element from one of the other list$_i$ parameters.

The = procedure's first argument is always an element of list$_1$; its second is an element of one of the other list$_i$. Elements that are repeated multiple times in the list$_1$ parameter will occur multiple times in the result. The order in which elements appear in the result is the same as they appear in list$_1$ – that is, `lset-difference` essentially filters list$_1$, without disarranging element order. The result may share a common tail with list$_1$. The dynamic order in which the applications of = are made is not specified. The procedure may check an element of list$_1$ for membership in every other list before proceeding to consider the next element of list$_1$, or it may completely compute the difference of list$_1$ and list$_2$ before proceeding to list$_3$, 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` = *list*$_1$ ...`)` → *list* list library procedure

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 list$_i$ 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` = *list*$_1$ *list*$_2$ *...*) → *list list* list library procedure

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 list$_1$; its second is an element of one of the other list$_i$.

Either of the answer lists may share a common tail with list$_1$. This operation essentially partitions list$_1$.

`(lset-union!` = *list*$_1$ *...*) → *list* list library procedure
`(lset-intersection!` = *list*$_1$ *list*$_2$ *...*) → *list* list library procedure
`(lset-difference!` = *list*$_1$ *list*$_2$ *...*) → *list* list library procedure
`(lset-xor!` = *list*$_1$ *...*) → *list* list library procedure
`(lset-diff+intersection!` = *list*$_1$ *list*$_2$ *...*) → *list list* list library procedure

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

Primitive side-effects

These two procedures are the primitive, R5RS side-effect operations on pairs.

`(set-car!` *pair object*) → *unspecified* list library procedure
`(set-cdr!` *pair object*) → *unspecified* list library procedure

[R5RS] 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*
```

References & links

TODO What of this should survive in the final document?

This document, in HTML: http://srfi.schemers.org/srfi-1/srfi-1.html
Source code for the reference implementation: http://srfi.schemers.org/srfi-1/srfi-1-reference.scm
Archive of SRFI-1 discussion-list email: http://srfi.schemers.org/srfi-1/mail-archive/maillist.html
SRFI web site: `http://srfi.schemers.org/`

[**CommonLisp**] *Common Lisp: the Language*
Guy L. Steele Jr. (editor).
Digital Press, Maynard, Mass., second edition 1990.
Available at `http://www.elwood.com/alu/table/references.htm#cltl2`.
The Common Lisp "HyperSpec," produced by Kent Pitman, is essentially the ANSI spec for Common Lisp: `http://www.harlequin.com/education/books/HyperSpec/`.

[**R5RS**] Revised^5 report on the algorithmic language Scheme.
R. Kelsey, W. Clinger, J. Rees (editors).
Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.
and ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998.
Available at `http://www.schemers.org/Documents/Standards/`.

2. Vector libraries

[**Red Edition item 2**] What vector library should R7RS-large provide?

This SRFI proposes a comprehensive library of vector operations accompanied by a freely available and complete reference implementation. The reference implementation is unencumbered by copyright, and useable with no modifications on any Scheme system that is R5RS-compliant. It also provides several hooks for implementation-specific optimization as well.

The names described in this section comprise the `(scheme vector)` library.

Procedure Index

Here is an index of the procedures provided by this package. Those marked by *italics* are also provided in R7RS-small.

Constructors `make-vector` `vector` `vector-unfold`
`vector-unfold-right` `vector-copy` `vector-reverse-copy`
`vector-append` `vector-concatenate` `vector-append-subvectors`

Predicates `vector?` `vector-empty?` `vector=`

Selectors `vector-ref` `vector-length`

Iteration `vector-fold` `vector-fold-right` `vector-map`
`vector-map!` `vector-for-each` `vector-count`
`vector-cumulate`

Searching `vector-index` `vector-index-right` `vector-skip`
`vector-skip-right` `vector-binary-search` `vector-any`
`vector-every` `vector-partition`

Mutators `vector-set!` `vector-swap!` `vector-fill!`
`vector-reverse!` `vector-copy!` `vector-reverse-copy!`
`vector-unfold!` `vector-unfold-right!`

Conversion `vector->list` `reverse-vector->list` `list->vector`
`reverse-list->vector` `vector->string` `string->vector`

Notation

TODO Unify with notation in the rest of the Report, move to a common section.

In this section containing specifications of procedures, the following notation is used to specify parameters and return values:

(f arg$_1$ arg$_2$ ...) -> something Indicates a function `f` takes the parameters `arg1 arg2` ... and returns a value of the type `something`. If `something` is `unspecified`, then `f` returns a single implementation-dependent value; this SRFI does not specify what it returns, and in order to write portable code, the return value should be ignored.

vec The argument in this place must be a vector, i.e. it must satisfy the predicate `vector?`.

i, j, start, size The argument in this place must be a exact nonnegative integer, i.e. it must satisfy the predicates `exact?`, `integer?` and either `zero?` or `positive?`. The third case of it indicates the index at which traversal begins; the fourth case of it indicates the size of a vector.

end The argument in this place must be a exact positive integer, i.e. it must satisfy the predicates `exact?`, `integer?` and `positive?`. This indicates the index directly before which traversal will stop — processing will occur until the the index of the vector is `end`. It is the closed right side of a range.

f The argument in this place must be a function of one or more arguments, which returns (except as noted otherwise) exactly one value.

pred? The argument in this place must be a function of one or more arguments that returns one value, which is treated as a boolean.

x, y, z, seed, knil, fill, key, value The argument in this place may be any Scheme value.

[**something**] Indicates that `something` is an optional argument; it needn't necessarily be applied. `Something` needn't necessarily be one thing; for example, this usage of it is perfectly valid: `[start [end]]` and is indeed used quite often.

something ... Indicates that zero or more `something`s are allowed to be arguments.

something$_1$ something$_2$ ... Indicates that at least one `something` must be arguments.

something$_1$ something$_2$ ... something$_n$ Exactly equivalent to the previous argument notation, but this also indicates that `n` will be used later in the procedure description.

It should be noted that all of the procedures that iterate across multiple vectors in parallel stop iterating and produce the final result when the end of the shortest vector is reached. The sole exception is `vector=`, which automatically returns `#f` if the vectors' lengths vary.

Constructors

`(make-vector` *size fill*`)` → *vector* vector library procedure
`(make-vector` *size*`)` → *vector* vector library procedure

[*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 ...*`)` → *vector* vector library procedure

[*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 ...*`)` → *vector* vector library procedure

] 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 `k`th 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:

```
(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`).

```
(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 …*`)` → *vector* vector library procedure

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*`)` → *vector* vector library procedure
`(vector-copy` *vec start*`)` → *vector* vector library procedure
`(vector-copy` *vec*`)` → *vector* vector library procedure

[*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*`)` → *vector* vector library procedure

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 …*`)` → *vector* vector library procedure

[*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*`)` → *vector* vector library procedure

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] ...*`)` → *vector* vector library procedure

TODO Check the signature on this one.

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)`

Predicates

`(vector?` *x*`)` → *boolean* vector library procedure

[*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*`)` → *boolean* vector library procedure

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 ...*`)` → *boolean* vector library procedure

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. Element comparison must be consistent with `eq`; that is, if `(eq? Ea Eb)` results in a true value, then `(elt=? Ea Eb)` must also result in a true value. This may be exploited to avoid unnecessary element comparisons. (The reference implementation does, but it does not consider the situation where `elt=?` is in fact itself `eq?` to avoid yet more unnecessary comparisons.) 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 or not literal vectors with the same external representation are `eq?`. `(vector= eq? (vector (vector 'a)) (vector (vector 'a)))`
`#f` `(vector= equal? (vector (vector 'a)) (vector (vector 'a)))`
`#t`

Selectors

TODO introduction

`(vector-ref` *vec i*`)` → *value* vector library procedure

[*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*`)` → *exact nonnegative integer* vector library procedure

[*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`

Iteration

TODO introduction

`(vector-fold` *kons knil vec*$_1$ *vec*$_2$ *...*`)` → *value* vector library procedure

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`
`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))`
duce a list of the reversed elements of `vec`.
`(vector-fold ( (tail elt) (cons elt tail)) '() vec)`
the number of even numbers in `vec`.
`(vector-fold ( (counter n) (if (even? n) (+ cour`

`(vector-fold-right` *kons knil vec*$_1$ *vec*$_2$ *...*`)` → *value* vector library procedure

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)`

`(vector-map` *f vec*$_1$ *vec*$_2$ *...*`)` → *vector* vector library procedure

[*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`
The dynamic order of application of `f` is unspecified. Examples: `(vector-map ( (x) (* x x))`
`#(1 4 9 16)` `(vector-map ( (x y) (* x y))` `(vector-unfold`
`#(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 vec*$_1$ *vec*$_2$ …) → *unspecified* vector library procedure

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 vec*$_1$ *vec*$_2$ …) → *unspecified* vector library procedure

[*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)`
`'#("foo" "bar" "baz" "quux" "zot"))`
Displays:

```
foo
bar
baz
quux
zot
```

(`vector-count` *pred? vec*$_1$ *vec*$_2$ …) → *exact nonnegative integer* vector library procedure

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*) → *vector* vector library procedure

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 *`new`*$_{i-1}$ and *`vec`*$_i$, 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 2016/9/2.

Example: `(vector-cumulate + 0 '#(3 1 4 1 5 9 2 5 6))`
`#(3 4 8 9 14 23 25 30 36)`

Searching

(`vector-index` *pred? vec*$_1$ *vec*$_2$ …) → *exact nonnegative integer or #f* vector library procedure

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? vec*$_1$ *vec*$_2$ *...*`)` → *exact nonnegative integer or #f* vector library procedure

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? vec*$_1$ *vec*$_2$ *...*`)` → *exact nonnegative integer or #f* vector library procedure

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: `(vector-index ( (x1 x2 ...) (not (pred? x1`
ample: `(vector-skip number? '#(1 2 a b 3 4 c d))`
`2`

`(vector-skip-right` *pred? vec*$_1$ *vec*$_2$ *...*`)` → *exact nonnegative integer or #f* vector library procedure

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`

`(vector-binary-search` *vec value cmp*`)` → *exact nonnegative integer or #f* vector library procedure

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? vec*$_1$ *vec*$_2$ *...*`)` → *value or #f* vector library procedure

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? vec*$_1$ *vec*$_2$ *...*`)` → *value or #f* vector library procedure

If, for every index `i` between 0 and the length of the shortest vector argument, the set of elements `(vector-ref vec1`
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*`)` → *vector* vector library procedure

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`.

Mutators

`(vector-set!` *vec i value*`)` → *unspecified* — vector library procedure

[*R7RS-small*] Assigns the contents of the location at `i` in `vec` to `value`.

`(vector-swap!` *vec i j*`)` → *unspecified* — vector library procedure

Swaps or exchanges the values of the locations in `vec` at `i` & `j`.

`(vector-fill!` *vec fill start end*`)` → *unspecified* — vector library procedure
`(vector-fill!` *vec fill start*`)` → *unspecified* — vector library procedure
`(vector-fill!` *vec fill*`)` → *unspecified* — vector library procedure

[*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*`)` → *unspecified* — vector library procedure
`(vector-reverse!` *vec start*`)` → *unspecified* — vector library procedure
`(vector-reverse!` *vec*`)` → *unspecified* — vector library procedure

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*`)` → *unspecified* — vector library procedure
`(vector-copy!` *to at from start*`)` → *unspecified* — vector library procedure
`(vector-copy!` *to at from*`)` → *unspecified* — vector library procedure

[*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*`)` → *unspecified* — vector library procedure
`(vector-reverse-copy!` *to at from start*`)` → *unspecified* — vector library procedure
`(vector-reverse-copy!` *to at from*`)` → *unspecified* — vector library procedure

Like `vector-copy!`, but the elements appear in `to` in reverse order. `(vector-reverse!` `target`

`(vector-unfold!` *f vec start end initial-seed …*`)` → *unspecified* — vector library procedure

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 …*`)` → *unspecified* — vector library procedure

Like `vector-unfold!`, but the elements are copied in reverse order into the vector *vec* starting at the index preceding *end*.

Conversion

`(vector->list`*vec start end*`)` → *proper-list* vector library procedure
`(vector->list`*vec start*`)` → *proper-list* vector library procedure
`(vector->list`*vec*`)` → *proper-list* vector library procedure

[*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*`)` → *proper-list* vector library procedure
`(reverse-vector->list` *vec start*`)` → *proper-list* vector library procedure
`(reverse-vector->list` *vec*`)` → *proper-list* vector library procedure

] Like `vector->list`, but the resulting list contains the elements in reverse of `vector`.

`(list->vector` *proper-list*`)` → *vector* vector library procedure

[*R7RS-small*] Creates a vector of elements from `proper-list`.

`(reverse-list->vector` *proper-list*`)` → *vector* vector library procedure

Like `list->vector`, but the resulting vector contains the elements in reverse of `proper-list`.

`(string->vector` *string start end*`)` → *vector* vector library procedure
`(string->vector` *string start*`)` → *vector* vector library procedure
`(string->vector` *string*`)` → *vector* vector library procedure

[*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*`)` → *string* vector library procedure
`(vector->string` *vec start*`)` → *string* vector library procedure
`(vector->string` *vec*`)` → *string* vector library procedure

[*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.

References

TODO move to consolidated reference list.

R5RS *R5RS: The Revised^5 Report on Scheme*
R. Kelsey, W. Clinger, J. Rees (editors).
Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998
and
ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998
Available at: `http://www.schemers.org/Documents/Standards/R5RS/`

R7RS-small *R7RS: The Revised^7 Report on Scheme*
A. Shinn et al. (editors).
Available at: `http://r7rs.org`

SRFI *SRFI: Scheme Request for Implementation*
The SRFI website can be found at: `http://srfi.schemers.org/`
The SRFIs mentioned in this document are described later.

SRFI 1 *SRFI 1: List Library*
A SRFI of list processing procedures, written by Olin Shivers.
Available at: `http://srfi.schemers.org/srfi-1/`

SRFI 13 *SRFI 13: String Library*
A SRFI of string processing procedures, written by Olin Shivers.
Available at: `http://srfi.schemers.org/srfi-13/`

SRFI 23 *SRFI 23: Error Reporting Mechanism*
A SRFI that defines a new primitive (`error`) for reporting that an error occurred, written by Stephan Houben.
Available at: `http://srfi.schemers.org/srfi-23/`

SRFI 43 *SRFI 43: Vector Library (draft)*
The direct predecessor of this SRFI, written by Taylor Campbell.
Available at: `http://srfi.schemers.org/srfi-43/`

Editor: Arthur A. Gleckler

String library

[**Red Edition item 3**] As no consensus was achieved on a string library, no string library is provided in the Red Edition; one might be provided later.

This section is left in as a placeholder, and will be removed later in the editing process.

3. Sorting library

[**Red Edition item 4**] What sorting library should R7RS-large provide?

> SRFI 132 is an edited version of the withdrawn SRFI 32, with the algorithm-specific procedures removed and a few additional procedures; it provides sortedness testing, stable and unstable sorting, merging, duplicate removal, median, and selection. R6RS provides only stable sorting and merging.

This SRFI describes the API for a full-featured sort toolkit. The names described in this section comprise the `(scheme sort)` library.

Procedure naming and functionality

Most of the procedures described below are variants of two basic operations: sorting and merging. These procedures are consistently named by composing a set of basic lexemes to indicate what they do.

Lexeme	Meaning
`vector`	The procedure operates upon vectors.
`list`	The procedure operates upon lists.
`stable`	This lexeme indicates that the sort is a stable one.
`sort`	The procedure sorts its input data set by some ordering function.
`merge`	The procedure merges two ordered data sets into a single ordered result.
`!`	Procedures that end in `!` are allowed, and sometimes required, to reuse their input storage to construct their answer.

Types of parameters and return values

In the procedures specified below:

- A *lis* parameter is a list.
- A *v* parameter is a vector.
- An = parameter is an equality predicate. See SRFI 128 for the requirements on equality predicates. Note that neither this SRFI nor its sample implementation depend on SRFI 128.

- A < parameter is an ordering predicate. See SRFI 128 for the requirements on ordering predicates. TODO use le? as a clearer name.

- A *start* parameter or *start* and *end* parameter pair are exact non-negative integers such that 0 <= *start* <= *end* <= `(vector-length` *v*`)`, where *v* is the related vector parameter. If not specified, they default to 0 and the length of the vector, respectively. They are interpreted to select the range [*start*, *end*), that is, all elements from index *start* (inclusive) up to, but not including, index *end*.

Passing values to procedures with these parameters that do not satisfy these constraints is an error.

If a procedure is said to return "an unspecified value", this means that nothing at all is said about what the procedure returns, except that it returns one value.

Predicates

TODO Change textless to le? throughout this section.

`(list-sorted?` < *lis*`)` → *boolean* sort library procedure
`(vector-sorted?` < *v start end*`)` → *boolean* sort library procedure
`(vector-sorted?` < *v start*`)` → *boolean* sort library procedure
`(vector-sorted?` < *v*`)` → *boolean* sort library procedure

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.

General sort procedures

These procedures provide basic sorting and merging functionality suitable for general programming. The procedures are named by their semantic properties, i.e., what they do to the data (sort, stable sort, and so forth).

`(list-sort` < *lis*`)` → *list???* sort library procedure
`(list-stable-sort` < *lis*`)` → *list???* sort library procedure

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???* sort library procedure
`(list-stable-sort!` < *lis*`)` → *list???* sort library procedure

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???* sort library procedure
`(vector-sort` < *v start*`)` → *vector???* sort library procedure
`(vector-sort` < *v end*`)` → *vector???* sort library procedure
`(vector-stable-sort` < *v start end*`)` → *vector???* sort library procedure

`(vector-stable-sort` $<$ *v start*`)` → *vector???* sort library procedure
`(vector-stable-sort` $<$ *v*`)` → *vector???* sort library procedure

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???* sort library procedure
`(vector-sort!` $<$ *v start*`)` → *vector???* sort library procedure
`(vector-sort!` $<$ *v*`)` → *vector???* sort library procedure
`(vector-stable-sort!` $<$ *v start end*`)` → *vector???* sort library procedure
`(vector-stable-sort!` $<$ *v start*`)` → *vector???* sort library procedure
`(vector-stable-sort!` $<$ *v*`)` → *vector???* sort library procedure

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.

Merge procedures

All four merge operations are stable: an element of the initial list lis_1 or vector v_1 will come before an equal-comparing element in the second list lis_2 or vector v_2 in the result.

`(list-merge` $<$ lis_1 lis_2`)` → *list???* sort library procedure

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!` $<$ lis_1 lis_2`)` → *list???* sort library procedure

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` $<$ v_1 v_2 $start_1$ end_1 $start_2$ end_2`)` → *vector???* sort library procedure
`(vector-merge` $<$ v_1 v_2 $start_1$ end_1 $start_2$`)` → *vector???* sort library procedure
`(vector-merge` $<$ v_1 v_2 $start_1$ end_1`)` → *vector???* sort library procedure
`(vector-merge` $<$ v_1 v_2 $start_1$`)` → *vector???* sort library procedure
`(vector-merge` $<$ v_1 v_2`)` → *vector???* sort library procedure

This procedure does not alter its inputs, and returns a newly allocated vector of length $(end_1 - start_1) + (end_2 - start_2)$.

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* $from_1$ $from_2$ *start* $start_1$ end_1 $start_2$ end_2`)` → *vector???* sort library procedure
`(vector-merge!` $<$ *to* $from_1$ $from_2$ *start* $start_1$ end_1 $start_2$`)` → *vector???* sort library procedure

`(vector-merge!` $<$ *to* $from_1$ $from_2$ *start* $start_1$ end_1`)` → *vector???* sort library procedure
`(vector-merge!` $<$ *to* $from_1$ $from_2$ *start* $start_1$`)` → *vector???* sort library procedure
`(vector-merge!` $<$ *to* $from_1$ $from_2$ *start*`)` → *vector???* sort library procedure
`(vector-merge!` $<$ *to* $from_1$ $from_2$`)` → *vector???* sort library procedure

This procedure writes its result into vector *to*, beginning at index *start*, for indices less than *end*, which is defined as *start* + (end_1 - $start_1$) + (end_2 - $start_2$). The target subvector *to*[*start*, *end*) may not overlap either of the source subvectors $from_1$[$start_1$, end_1] and $from_2$[$start_2$, end_2]. 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.

Deleting duplicate neighbors

These procedures delete adjacent duplicate elements from a list or a vector, using a given element-equality procedure. The first/leftmost element of a run of equal elements is the one that survives. The list or vector is not otherwise disordered.

These procedures are linear time — much faster than the O(n^2) general duplicate-element deletion procedures that do not assume any "bunching" of elements provided by SRFI 1. If you want to delete duplicate elements from a large list or vector, sort the elements to bring equal items together, then use one of these procedures, for a total time of O(n lg n).

The equality procedure is always invoked as `(= x y)`, where *x* comes before *y* in the containing list or vector.

`(list-delete-neighbor-dups` = *lis*`)` → *list???* sort library procedure

This procedure does not alter its input list, but its result may share storage with the input list.

`(list-delete-neighbor-dups!` = *lis*`)` → *list???* sort library procedure

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*`)` → *vector???* sort library procedure
`(vector-delete-neighbor-dups` = *v start*`)` → *vector???* sort library procedure
`(vector-delete-neighbor-dups` = *v*`)` → *vector???* sort library procedure

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*`)` → *vector???* sort library procedure
`(vector-delete-neighbor-dups!` = *v start*`)` → *vector???* sort library procedure
`(vector-delete-neighbor-dups!` = *v*`)` → *vector???* sort library procedure

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))
```

Finding the median

These procedures do not have SRFI 32 counterparts. They find the median element of a vector after sorting it in accordance with an ordering procedure. If the number of elements in `v` is odd, the middlemost element of the sorted result is returned. If the number of elements is zero, `knil` is returned. Otherwise, `mean` is applied to the two middlemost elements in the order in which they appear in `v`, and whatever it returns is returned. If `mean` is omitted, then the default mean procedure is `(lambda (a b) (/ (+ a b) 2)`, but this procedure is applicable to non-numeric values as well.

`(vector-find-median` $<$ *v knil mean*`)` → *value???* sort library procedure
`(vector-find-median` $<$ *v knil*`)` → *value???* sort library procedure

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*`)` → *value???* sort library procedure
`(vector-find-median!` $<$ *v knil*`)` → *value???* sort library procedure

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. TODO Should that be $O(n \lg n)$? And aren't there median algorithms that run in $O(n)$ time?

Selection

These procedures do not have SRFI 32 counterparts.

`(vector-select!` $<$ *v k start end*`)` → *value???* sort library procedure
`(vector-select!` $<$ *v k start*`)` → *value???* sort library procedure
`(vector-select!` $<$ *v k*`)` → *value???* sort library procedure

This procedure returns the *k*th 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*`)` → *unspecified* sort library procedure
`(vector-separate!` $<$ *v k start*`)` → *unspecified* sort library procedure
`(vector-separate!` $<$ *v k*`)` → *unspecified* sort library procedure

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.

Editor: Arthur A. Gleckler

4. Sets and maps

General sets and bags

[**Red Edition item 5**] What general set and bag library should R7RS-large provide?

> SRFI 113 provides linear-update sets and bags, which can be mutable or immutable according to the implementer's preference. Since the interface of SRFI 113 depends on SRFI 114, which has been replaced by SRFI 128, voting for SRFI 113 will cause your vote on comparator libraries to be automatically changed to SRFI 128.
>
> TODO Incorporate these post-finalization notes into the text, if not already done. Maybe comparators should come before this SRFI?

> **Post-finalization note 1**: Because SRFI 114 has been deprecated by SRFI 128, it is recommended that implementers make use of SRFI 128 rather than SRFI 114 comparators where comparators are specified in this SRFI. Specifically, the procedures `set`, `bag`, `set-unfold`, `bag-unfold`, `set-map`, `list->set`, `list->bag`, and `alist->bag`, should accept SRFI 128 rather than SRFI 114 comparators as arguments. By the same token, the results of `set-element-comparator` and `bag-element-comparator`, as well as the values of `set-comparator` and `bag-comparator`, should be SRFI 128 comparators. The sample implementation has been updated to depend on SRFI 128 rather than SRFI 114.
>
> **Post-finalization note 2**: The order of arguments of `set-map` and `set-unfold` are not consistent with those of `hash-table-map` and `hash-table-unfold` in SRFI 125. Both SRFI 113 and SRFI 125 are going to be part of R7RS large (also known as the Red Edition), and the plan is to make them consistent there, following the order in SRFI 125. Furthermore, SRFI 146 (in draft status at the time of this writing) uses the order of SRFI 125. Since this problem was discovered after SRFI 113 was finalized, and it wasn't an error in the SRFI, it's too late to fix it here. However, John Cowan, the author, encourages implementers to consider adopting the order of SRFI 125. Thanks to Marc Nieper-Wißkirchen for reporting this mismatch, and for providing a version of SRFI 113 with the recommended argument order in the implementation of SRFI 146.

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.

The names described in this section comprise the `(scheme set)` library.

Sets and bags are mutually disjoint, and disjoint from other types of Scheme objects.

It is an error for any procedure defined in this SRFI to be invoked on sets or bags with distinct comparators (in the sense of `eq?`).

It is an error to mutate any object while it is contained in a set or bag.

It is an error to add an object to a set or bag which does not satisfy the type test predicate of the comparator.

It is an error to add or remove an object for a set or a bag while iterating over it.

The procedures of this SRFI, by default, are "pure functional": they do not alter their parameters. However, this SRFI also defines "linear-update" procedures, all of whose names end in `!`. They have hybrid pure-functional/side-effecting semantics: they are allowed, but not required, to side-effect one of their parameters in order to construct their result. An implementation may legally implement these procedures as pure, side-effect-free functions, or it may implement them using side effects, depending upon the details of what is the most efficient or simple to implement in terms of the underlying representation.

It is an error to rely upon these procedures working by side effect. For example, this is not guaranteed to work:

```
(let* ((set1 (set 'a 'b 'c))      ; set1 = {a,b,c}.
       (set2 (set-adjoin! set1 'd)))   ; Add d to {a,b,c}.
  set1) ; Could be either {a,b,c} or {a,b,c,d}.
```

However, this is well-defined:

```
(let ((set1 (set 'a 'b 'c)))
  (set-adjoin! set1 'd)) ; Add d to {a,b,c}.
```

So clients of these procedures write in a functional style, but must additionally be sure that, when the procedure is called, there are no other live pointers to the potentially-modified set or bag (hence the term "linear update").

There are two benefits to this convention:

- Implementations are free to provide the most efficient possible implementation, either functional or side-effecting.
- Programmers may nonetheless continue to assume that sets are purely functional data structures: they may be reliably shared without needing to be copied, uniquified, and so forth.

In practice, these procedures are most useful for efficiently constructing sets and bags in a side-effecting manner, in some limited local context, before passing the set or bag outside the local construction scope to be used in a functional manner.

Scheme provides no assistance in checking the linearity of the potentially side-effected parameters passed to these functions — there's no linear type checker or run-time mechanism for detecting violations.

Note that if an implementation uses no side effects at all, it is allowed to return existing sets and bags rather than newly allocated ones, even where this SRFI explicitly says otherwise.

Comparator restrictions

Implementations of this SRFI are allowed to place restrictions on the comparators that the procedures accept. In particular, an implementation may require comparators to provide a comparison procedure. Alternatively, an implementation may require comparators to provide a hash function, unless the equality predicate of the comparator is `eq?`, `eqv?`, `equal?`, `string=?`, or `string-ci=?`. Implementations must not require the provision of both a comparison procedure and a hash function.

Index

Set procedures

(`set` *comparator element …*) → *set* set library procedure

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*) → *set* set library procedure

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*`)` → *boolean* set library procedure

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

`(set-contains?` *set element*`)` → *boolean* set library procedure

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

`(set-empty?` *set*`)` → *boolean* set library procedure

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

`(set-disjoint?` *set*$_1$ *set*$_2$`)` → *boolean* set library procedure

Returns `#t` if *set*$_1$ and *set*$_2$ have no elements in common and `#f` otherwise.

Accessors

`(set-member` *set element default*`)` → *value???* set library procedure

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*`)` → *comparator* set library procedure

Returns the comparator used to compare the elements of *set*.

Updaters

`(set-adjoin` *set element* …`)` → *set???* set library procedure

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* …`)` → *set???* set library procedure

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*`)` → *set???* set library procedure

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*`)` → *set???* set library procedure

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

`(set-delete` *set element* …`)` → *set???* set library procedure
`(set-delete!` *set element* …`)` → *set???* set library procedure

`(set-delete-all` *set element-list*`)` → *set???* set library procedure
`(set-delete-all!` *set element-list*`)` → *set???* set library procedure

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*`)` → *set??? obj??* set library procedure

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*.

The whole set

`(set-size` *set*`)` → *integer* set library procedure

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

`(set-find` *predicate set failure*`)` → *object???* set library procedure

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*`)` → *integer* set library procedure

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

`(set-any?` *predicate set*`)` → *boolean???* set library procedure

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*`)` → *boolean???* set library procedure

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.

Mapping and folding

(`set-map` *comparator proc set*) → *boolean???* set library procedure

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*) → *unspecified???* set library procedure

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

(`set-fold` *proc nil set*) → *object???* set library procedure

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*) → *set???* set library procedure

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

(`set-filter!` *predicate set*) → *set???* set library procedure

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

(`set-remove` *predicate set*) → *set???* set library procedure

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*) → *set???* set library procedure

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

(`set-partition` *predicate set*) → *set??? set???* set library procedure

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*) → *set??? set???* set library procedure

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

Copying and conversion

`(set-copy` *set*`)` → *set???* set library procedure

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

`(set->list` *set*`)` → *list???* set library procedure

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

`(list->set` *comparator list*`)` → *set???* set library procedure

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 listprocedureset???

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

Subsets

Note: The following three predicates do not obey the trichotomy law and therefore do not constitute a total order on sets.

TODO Explain how elements are compared. Via their comparators?

`(set=?` *set*$_1$ *set*$_2$ ...`)` → *boolean???* set library procedure

Returns `#t` if each *set* contains the same elements.

`(set<?)` → *procedure* set library set$_1$ set$_2$ ...
boolean???

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

`(set>?`*set*$_1$ *set*$_2$ ...`)` → *boolean???* set library procedure

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

`(set<=?` *set*$_1$ *set*$_2$ ...`)` → *boolean???* set library procedure

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

`(set>=?` *set*$_1$ *set*$_2$ ...`)` → *boolean???* set library procedure

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

Set theory operations

`(set-union` *set*$_1$ *set*$_2$ … `)` → *set???* set library procedure
`(set-intersection` *set*$_1$ *set*$_2$ … `)` → *set???* set library procedure
`(set-difference` *set*$_1$ *set*$_2$ … `)` → *set???* set library procedure
`(set-xor` *set*$_1$ *set*$_2$`)` → *set???* set library procedure

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!` *set*$_1$ *set*$_2$ … `)` → *set???* set library procedure
`(set-intersection!` *set*$_1$ *set*$_2$ … `)` → *set???* set library procedure
`(set-difference!` *set*$_1$ *set*$_2$ … `)` → *set???* set library procedure
`(set-xor!` *set*$_1$ *set*$_2$`)` → *set???* set library procedure

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 procedures

Bags are like sets, but can contain the same object more than once. However, if two elements that are the same in the sense of the equality predicate, but not in the sense of `eqv?`, are both included, it is not guaranteed that they will remain distinct when retrieved from the bag. It is an error for a single procedure to be invoked on bags with different comparators.

TODO This must be rewritten in standard Scheme entry format.

The procedures for creating and manipulating bags are the same as those for sets, except that `set` is replaced by `bag` in their names, and that adjoining an element to a bag is effective even if the bag already contains the element. If two elements in a bag are the same in the sense of the bag's comparator, the implementation may in fact store just one of them.

The `bag-union`, `bag-intersection`, `bag-difference`, and `bag-xor` procedures (and their linear update analogues) behave as follows when both bags contain elements that are equal in the sense of the bags' comparator:

- For `bag-union`, the number of equal elements in the result is the largest number of equal elements in any of the original bags.
- For `bag-intersection`, the number of equal elements in the result is the smallest number of equal elements in any of the original bags.
- For `bag-difference`, the number of equal elements in the result is the number of equal elements in the first bag, minus the number of elements in the other bags (but not less than zero).
- For `bag-xor`, the number of equal elements in the result is the absolute value of the difference between the number of equal elements in the first and second bags.

Additional bag procedures

`(bag-sum` *set*$_1$ *set*$_2$ … `)` → *bag???* set library procedure
`(bag-sum!` *bag*$_1$ *bag*$_2$ … `)` → *bag???* set library procedure

TODO are those really `set` arguments in `bag-sum`?

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???* set library procedure
(`bag-product!` *n bag*) → *bag???* set library procedure

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*) → *integer???* set library procedure

Returns the number of unique elements of *bag* TODO as an exact integer??? .

(`bag-element-count` *bag element*) → *integer???* set library procedure

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

(`bag-for-each-unique` *proc bag*) → *unspecified???* set library procedure

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*) → *value???* set library procedure

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???* set library procedure
(`bag-decrement!` *bag element count*) → *bag???* set library procedure

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???* set library procedure
(`set->bag` *set*) → *bag???* set library procedure
(`set->bag!` *bag set*) → *bag???* set library procedure

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???* set library procedure
(`alist->bag` *comparator alist*) → *bag???* set library procedure

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

TODO This section must be expanded.

The following comparators are used to compare sets or bags, and allow sets of sets, bags of sets, etc.

`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.

Editor: Mike Sperber

Character sets

[**Red Edition item 6**] What character set library should R7RS-large provide?

> SRFI 14 provides linear-update sets and bags, which can be mutable or immutable according to the implementer's preference. The contents are limited to characters.

Abstract

The ability to efficiently represent and manipulate sets of characters is an unglamorous but very useful capability for text-processing code – one that tends to pop up in the definitions of other libraries. Hence it is useful to specify a general substrate for this functionality early. This SRFI defines a general library that provides this functionality. It is accompanied by a reference implementation for the spec. The reference implementation is fairly efficient, straightforwardly portable, and has a "free software" copyright. The implementation is tuned for "small" 7 or 8 bit character types, such as ASCII or Latin-1; the data structures and algorithms would have to be altered for larger 16 or 32 bit character types such as Unicode – however, the specs have been carefully designed with these larger character types in mind. Several forthcoming SRFIs can be defined in terms of this one:

- string library
- delimited input procedures (*e.g.*, `read-line`)
- regular expressions

TODO is there any "tuning" needed of the spec itself for Unicode, as opposed to the reference implementation? I think not, but the above paragraph did give me pause. Also, references throughout the document are to older versions of the Unicode standard. I would hope we would in this document be able to limit ourselves to referring only to the Unicode character classes, which should only change in new editions of the standard by adding new characters to existing classes. If we're not careful, this spec will need to be revised for each new edition of the standard.

TODO the "forthcoming" SRFIs listed above do not seem to be included in the Red Edition.

TODO The library name is `(scheme charset)`, but the names are prefixed by `char-set-...`. Recipe for disaster?

The names described in this section comprise the `(scheme charset)` library.

Variable Index

Here is the complete set of bindings – procedural and otherwise – exported by this library. In a Scheme system that has a module or package system, these procedures should be contained in a module named "char-set-lib".

Predicates & comparison `char-set? char-set= char-set<= char-set-hash`

Iterating over character sets `char-set-cursor char-set-ref`
```
char-set-cursor-next end-of-char-set?
char-set-fold char-set-unfold char-set-unfold!
char-set-for-each char-set-map
```

Creating character sets `char-set-copy char-set`

```
list->char-set  string->char-set
list->char-set! string->char-set!

char-set-filter  ucs-range->char-set
char-set-filter! ucs-range->char-set!

->char-set
```

Querying character sets `char-set->list char-set->string`

```
char-set-size char-set-count
char-set-contains?
char-set-every char-set-any
```

Character-set algebra `char-set-adjoin  char-set-delete`

```
char-set-adjoin! char-set-delete!

char-set-complement  char-set-union
char-set-complement! char-set-union!
char-set-intersection
char-set-intersection!

char-set-difference  char-set-xor
char-set-difference! char-set-xor!
char-set-diff+intersection
char-set-diff+intersection!
```

Standard character sets `char-set:lower-case  char-set:upper-case`

```
char-set:title-case
char-set:letter      char-set:digit
char-set:letter+digit
char-set:graphic     char-set:printing
char-set:whitespace
char-set:iso-control char-set:punctuation
char-set:symbol
char-set:hex-digit   char-set:blank
char-set:ascii
char-set:empty       char-set:full
```

Specification

In the following procedure specifications:

- A cs parameter is a character set.
- An s parameter is a string.
- A char parameter is a character.
- A char-list parameter is a list of characters.
- A pred parameter is a unary character predicate procedure, returning a true/false value when applied to a character.
- An obj parameter may be any value at all.

Passing values to procedures with these parameters that do not satisfy these types is an error.

Unless otherwise noted in the specification of a procedure, procedures always return character sets that are distinct (from the point of view of the linear-update operations) from the parameter character sets. For example, `char-set-adjoin` is guaranteed to provide a fresh character set, even if it is not given any character parameters.

Parameters given in square brackets are optional. Unless otherwise noted in the text describing the procedure, any prefix of these optional parameters may be supplied, from zero arguments to the full list. When a procedure returns multiple values, this is shown by listing the return values in square brackets, as well. So, for example, the procedure with signature

```
halts? f [x init-store] -> [boolean integer]
```

would take one (f), two (f, x) or three (f, x, init-store) input parameters, and return two values, a boolean and an integer.

TODO I'll be changing the entries so they match the R7RS entry format, i.e., multiple prototypes for optional arguments.

A parameter followed by "`...`" means zero-or-more elements. So the procedure with the signature

```
sum-squares x ...  -> number
```

takes zero or more arguments (x …), while the procedure with signature

```
spell-check doc dict1 dict2 ... -> string-list
```

takes two required parameters (doc and dict_1) and zero or more optional parameters (dict_2 …).

General procedures

`(char-set?` *obj*`)` → *boolean* charset library ???

Is the object obj a character set?

`(char-set=` *cs_1* *…*`)` → *boolean* charset library ???

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<=` *cs_1* *…*`)` → *boolean*

charset library ???

Returns true if every character set cs_i is a subset of character set $\text{cs}_{\text{i}+1}$.

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*`)` → *integer* charset library ???
`(char-set-hash` *cs*`)` → *integer* charset library ???

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.

Iterating over character sets

`(char-set-cursor` *cset*`)` → *cursor* charset library ???
`(char-set-ref` *cset cursor*`)` → *char* charset library ???
`(char-set-cursor-next` *cset cursor*`)` → *cursor* charset library ???
`(end-of-char-set?` *cursor*`)` → *boolean* charset library ???

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*`)` → *object* charset library ???

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* charset library ???
`(char-set-unfold` *f p g seed*`)` → *char-set* charset library ???
`(char-set-unfold!` *f p g seed base-cs* `)` → *char-set* charset library ???

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), (g^2 seed), (g^3 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*`)` → *unspecified* charset library ???

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* charset library ???

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)
```

Creating character sets

`(char-set-copy` *cs*`)` → *char-set* charset library ???

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` *$char_1$ …*`)` → *char-set* charset library ???

Return a character set containing the given characters.

`(list->char-set` *char-list base-cs*`)` → *char-set* charset library procedure
`(list->char-set` *char-list*`)` → *char-set* charset library procedure
`(list->char-set!` *char-list base-cs*`)` → *char-set* charset library procedure

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*`)` → *char-set* charset library procedure
`(string->char-set` *s*`)` → *char-set* charset library procedure
`(string->char-set!` *s base-cs*`)` → *char-set* charset library procedure

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* charset library procedure
`(char-set-filter` *pred cs*`)` → *char-set* charset library procedure
`(char-set-filter!` *pred cs base-cs*`)` → *char-set* charset library procedure

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*`)` → *char-set* charset library procedure
`(ucs-range->char-set` *lower upper*`)` → *char-set* charset library procedure
`(ucs-range->char-set!` *lower upper error? base-cs*`)` → *char-set* charset library procedure

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*`)` → *char-set* charset library procedure

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.

Querying character sets

`(char-set-size` *cs*`)` → *integer* charset library procedure

Returns the number of elements in character set cs.

`(char-set-count` *pred cs*`)` → *integer* charset library procedure

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*`)` → *character-list* charset library procedure

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*`)` → *string* charset library procedure

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*`)` → *boolean* charset library procedure

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*`)` → *boolean* charset library procedure
`(char-set-any` *pred cs*`)` → *boolean* charset library procedure

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)
```

Character-set algebra

`(char-set-adjoin` *cs* $char_1$ …`)` → *char-set* charset library procedure
`(char-set-delete` *cs* $char_1$ …`)` → *char-set* charset library procedure

Add/delete the $char_i$ characters to/from character set cs.

`(char-set-adjoin!` *cs* $char_1$ …`)` → *char-set* charset library procedure
`(char-set-delete!` *cs* $char_1$ …`)` → *char-set* charset library procedure

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

`(char-set-complement` *cs*`)` → *char-set* charset library procedure
`(char-set-union` cs_1 …`)` → *char-set* charset library procedure
`(char-set-intersection` cs_1 …`)` → *char-set* charset library procedure
`(char-set-difference` cs_1 cs_2 …`)` → *char-set* charset library procedure
`(char-set-xor` cs_1 …`)` → *char-set* charset library procedure
`(char-set-diff+intersection` cs_1 cs_2 …`)` → *char-set char-set* charset library procedure

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* charset library procedure
(`char-set-union!` cs_1 cs_2 …) → *char-set* charset library procedure
(`char-set-intersection!` cs_1 cs_2 …) → *char-set* charset library procedure
(`char-set-difference!` cs_1 cs_2 …) → *char-set* charset library procedure
(`char-set-xor!` cs_1 cs_2 …) → *char-set* charset library procedure
(`char-set-diff+intersection!` cs_1 cs_2 cs_3 …) → *char-set char-set* charset library procedure

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, cs_1 and cs_2.

TODO why does diff+intersection have cs_3?

Standard character sets

TODO Convert to entry format?

Several character sets are predefined for convenience:

`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

Note that there may be characters in `char-set:letter` that are neither upper or lower case—this might occur in implementations that use a character type richer than ASCII, such as Unicode. A "graphic character" is one that would put ink on your page. While the exact composition of these sets may vary depending upon the character type provided by the underlying Scheme system, here are the definitions for some of the sets in an ASCII implementation:

`char-set:lower-case`	a-z
`char-set:upper-case`	A-Z
`char-set:letter`	A-Z and a-z
`char-set:digit`	0123456789
`char-set:punctuation`	!"#%&'()*,-./:;?@[\]_{}
`char-set:symbol`	$+<=>^`|~
`char-set:whitespace`	Space, newline, tab, form feed, vertical tab, carriage return
`char-set:blank`	Space and tab
`char-set:graphic`	letter + digit + punctuation + symbol
`char-set:printing`	graphic + whitespace
`char-set:iso-control`	ASCII 0-31 and 127

Note that the existence of the `char-set:ascii` set implies that the underlying character set is required to be at least as rich as ASCII (including ASCII's control characters).

Rationale: The name choices reflect a shift from the older "alphabetic/numeric" terms found in R5RS and Posix to newer, Unicode-influenced "letter/digit" lexemes.

Unicode, Latin-1 and ASCII definitions of the standard character sets

TODO Does this belong here? Would it not be better to refer to external Java documentation, and/or Unicode standard?

In Unicode Scheme implementations, the base character sets are compatible with Java's Unicode specifications. For ASCII or Latin-1, we simply restrict the Unicode set specifications to their first 128 or 256 codes, respectively. Scheme implementations that are not based on ASCII, Latin-1 or Unicode should attempt to preserve the sense or spirit of these definitions.

The following descriptions frequently make reference to the "Unicode character database." This is a file, available at URL

`ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.txt`

Each line contains a description of a Unicode character. The first semicolon-delimited field of the line gives the hex value of the character's code; the second field gives the name of the character, and the third field gives a two-letter category. Other fields give simple 1-1 case-mappings for the character and other information; see

`ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.html`

for further description of the file's format. Note in particular the two-letter category specified in the the third field, which is referenced frequently in the descriptions below.

char-set:lower-casechar-set:lower-case

For Unicode, we follow Java's specification: a character is lowercase if

- it is not in the range [U+2000,U+2FFF], and
- the Unicode attribute table does not give a lowercase mapping for it, and
- at least one of the following is true:
  - the Unicode attribute table gives a mapping to uppercase for the character, or
  - the name for the character in the Unicode attribute table contains the words "SMALL LETTER" or "SMALL LIGATURE".

The lower-case ASCII characters are

abcdefghijklmnopqrstuvwxyz

Latin-1 adds another 33 lower-case characters to the ASCII set:

00B5	MICRO SIGN
00DF	LATIN SMALL LETTER SHARP S
00E0	LATIN SMALL LETTER A WITH GRAVE
00E1	LATIN SMALL LETTER A WITH ACUTE
00E2	LATIN SMALL LETTER A WITH CIRCUMFLEX
00E3	LATIN SMALL LETTER A WITH TILDE
00E4	LATIN SMALL LETTER A WITH DIAERESIS
00E5	LATIN SMALL LETTER A WITH RING ABOVE
00E6	LATIN SMALL LETTER AE
00E7	LATIN SMALL LETTER C WITH CEDILLA
00E8	LATIN SMALL LETTER E WITH GRAVE
00E9	LATIN SMALL LETTER E WITH ACUTE
00EA	LATIN SMALL LETTER E WITH CIRCUMFLEX
00EB	LATIN SMALL LETTER E WITH DIAERESIS
00EC	LATIN SMALL LETTER I WITH GRAVE
00ED	LATIN SMALL LETTER I WITH ACUTE
00EE	LATIN SMALL LETTER I WITH CIRCUMFLEX
00EF	LATIN SMALL LETTER I WITH DIAERESIS
00F0	LATIN SMALL LETTER ETH
00F1	LATIN SMALL LETTER N WITH TILDE
00F2	LATIN SMALL LETTER O WITH GRAVE
00F3	LATIN SMALL LETTER O WITH ACUTE
00F4	LATIN SMALL LETTER O WITH CIRCUMFLEX
00F5	LATIN SMALL LETTER O WITH TILDE
00F6	LATIN SMALL LETTER O WITH DIAERESIS
00F8	LATIN SMALL LETTER O WITH STROKE
00F9	LATIN SMALL LETTER U WITH GRAVE
00FA	LATIN SMALL LETTER U WITH ACUTE
00FB	LATIN SMALL LETTER U WITH CIRCUMFLEX
00FC	LATIN SMALL LETTER U WITH DIAERESIS
00FD	LATIN SMALL LETTER Y WITH ACUTE
00FE	LATIN SMALL LETTER THORN
00FF	LATIN SMALL LETTER Y WITH DIAERESIS

Note that three of these have no corresponding Latin-1 upper-case character:

00B5	MICRO SIGN
00DF	LATIN SMALL LETTER SHARP S
00FF	LATIN SMALL LETTER Y WITH DIAERESIS

(The compatibility micro character uppercases to the non-Latin-1 Greek capital mu; the German sharp s character uppercases to the pair of characters "SS," and the capital y-with-diaeresis is non-Latin-1.)

(Note that the Java spec for lowercase characters given at

`http://java.sun.com/docs/books/jls/html/javalang.doc4.html#14345`

is inconsistent. U+00B5 MICRO SIGN fulfills the requirements for a lower-case character (as of Unicode 3.0), but is not given in the numeric list of lower-case character codes.)

(Note that the Java spec for `isLowerCase()` given at

`http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Character.html#isLowerCase(char)`

gives three mutually inconsistent definitions of "lower case." The first is the definition used in this SRFI. Following text says "A character is considered to be lowercase if and only if it is specified to be lowercase by the Unicode 2.0 standard (category Ll in the Unicode specification data file)." The former spec excludes U+00AA FEMININE ORDINAL INDICATOR and U+00BA MASCULINE ORDINAL INDICATOR; the later spec includes them. Finally, the spec enumerates a list of characters in the Latin-1 subset; this list excludes U+00B5 MICRO SIGN, which is included in both of the previous specs.)

char-set:upper-casechar-set:upper-case

For Unicode, we follow Java's specification: a character is uppercase if

- it is not in the range [U+2000,U+2FFF], and
- the Unicode attribute table does not give an uppercase mapping for it (this excludes titlecase characters), and
- at least one of the following is true:
  - the Unicode attribute table gives a mapping to lowercase for the character, or
  - the name for the character in the Unicode attribute table contains the words "CAPITAL LETTER" or "CAPITAL LIGATURE".

The upper-case ASCII characters are

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Latin-1 adds another 30 upper-case characters to the ASCII set:

00C0	LATIN CAPITAL LETTER A WITH GRAVE
00C1	LATIN CAPITAL LETTER A WITH ACUTE
00C2	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
00C3	LATIN CAPITAL LETTER A WITH TILDE
00C4	LATIN CAPITAL LETTER A WITH DIAERESIS
00C5	LATIN CAPITAL LETTER A WITH RING ABOVE
00C6	LATIN CAPITAL LETTER AE
00C7	LATIN CAPITAL LETTER C WITH CEDILLA
00C8	LATIN CAPITAL LETTER E WITH GRAVE
00C9	LATIN CAPITAL LETTER E WITH ACUTE
00CA	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
00CB	LATIN CAPITAL LETTER E WITH DIAERESIS
00CC	LATIN CAPITAL LETTER I WITH GRAVE
00CD	LATIN CAPITAL LETTER I WITH ACUTE
00CE	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
00CF	LATIN CAPITAL LETTER I WITH DIAERESIS
00D0	LATIN CAPITAL LETTER ETH
00D1	LATIN CAPITAL LETTER N WITH TILDE
00D2	LATIN CAPITAL LETTER O WITH GRAVE
00D3	LATIN CAPITAL LETTER O WITH ACUTE
00D4	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
00D5	LATIN CAPITAL LETTER O WITH TILDE
00D6	LATIN CAPITAL LETTER O WITH DIAERESIS
00D8	LATIN CAPITAL LETTER O WITH STROKE
00D9	LATIN CAPITAL LETTER U WITH GRAVE
00DA	LATIN CAPITAL LETTER U WITH ACUTE
00DB	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
00DC	LATIN CAPITAL LETTER U WITH DIAERESIS
00DD	LATIN CAPITAL LETTER Y WITH ACUTE
00DE	LATIN CAPITAL LETTER THORN

char-set:title-casechar-set:title-case

In Unicode, a character is titlecase if it has the category Lt in the character attribute database. There are very few of these characters; here is the entire 31-character list as of Unicode 3.0:

01C5	LATIN CAPITAL LETTER D WITH SMALL LETTER Z WITH CARON
01C8	LATIN CAPITAL LETTER L WITH SMALL LETTER J
01CB	LATIN CAPITAL LETTER N WITH SMALL LETTER J
01F2	LATIN CAPITAL LETTER D WITH SMALL LETTER Z
1F88	GREEK CAPITAL LETTER ALPHA WITH PSILI AND PROSGEGRAMMENI
1F89	GREEK CAPITAL LETTER ALPHA WITH DASIA AND PROSGEGRAMMENI
1F8A	GREEK CAPITAL LETTER ALPHA WITH PSILI AND VARIA AND PROSGEGRAMMENI
1F8B	GREEK CAPITAL LETTER ALPHA WITH DASIA AND VARIA AND PROSGEGRAMMENI
1F8C	GREEK CAPITAL LETTER ALPHA WITH PSILI AND OXIA AND PROSGEGRAMMENI
1F8D	GREEK CAPITAL LETTER ALPHA WITH DASIA AND OXIA AND PROSGEGRAMMENI
1F8E	GREEK CAPITAL LETTER ALPHA WITH PSILI AND PERISPOMENI AND PROSGEGRAMMENI
1F8F	GREEK CAPITAL LETTER ALPHA WITH DASIA AND PERISPOMENI AND PROSGEGRAMMENI
1F98	GREEK CAPITAL LETTER ETA WITH PSILI AND PROSGEGRAMMENI
1F99	GREEK CAPITAL LETTER ETA WITH DASIA AND PROSGEGRAMMENI
1F9A	GREEK CAPITAL LETTER ETA WITH PSILI AND VARIA AND PROSGEGRAMMENI
1F9B	GREEK CAPITAL LETTER ETA WITH DASIA AND VARIA AND PROSGEGRAMMENI
1F9C	GREEK CAPITAL LETTER ETA WITH PSILI AND OXIA AND PROSGEGRAMMENI
1F9D	GREEK CAPITAL LETTER ETA WITH DASIA AND OXIA AND PROSGEGRAMMENI
1F9E	GREEK CAPITAL LETTER ETA WITH PSILI AND PERISPOMENI AND PROSGEGRAMMENI
1F9F	GREEK CAPITAL LETTER ETA WITH DASIA AND PERISPOMENI AND PROSGEGRAMMENI
1FA8	GREEK CAPITAL LETTER OMEGA WITH PSILI AND PROSGEGRAMMENI
1FA9	GREEK CAPITAL LETTER OMEGA WITH DASIA AND PROSGEGRAMMENI
1FAA	GREEK CAPITAL LETTER OMEGA WITH PSILI AND VARIA AND PROSGEGRAMMENI
1FAB	GREEK CAPITAL LETTER OMEGA WITH DASIA AND VARIA AND PROSGEGRAMMENI
1FAC	GREEK CAPITAL LETTER OMEGA WITH PSILI AND OXIA AND PROSGEGRAMMENI
1FAD	GREEK CAPITAL LETTER OMEGA WITH DASIA AND OXIA AND PROSGEGRAMMENI
1FAE	GREEK CAPITAL LETTER OMEGA WITH PSILI AND PERISPOMENI AND PROSGEGRAMMENI
1FAF	GREEK CAPITAL LETTER OMEGA WITH DASIA AND PERISPOMENI AND PROSGEGRAMMENI
1FBC	GREEK CAPITAL LETTER ALPHA WITH PROSGEGRAMMENI
1FCC	GREEK CAPITAL LETTER ETA WITH PROSGEGRAMMENI
1FFC	GREEK CAPITAL LETTER OMEGA WITH PROSGEGRAMMENI

There are no ASCII or Latin-1 titlecase characters.

char-set:letterchar-set:letter

In Unicode, a letter is any character with one of the letter categories (Lu, Ll, Lt, Lm, Lo) in the Unicode character database.

There are 52 ASCII letters

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ

There are 117 Latin-1 letters. These are the 115 characters that are members of the Latin-1 `char-set:lower-case` and `char-set:upper-case` sets, plus

00AA	FEMININE ORDINAL INDICATOR
00BA	MASCULINE ORDINAL INDICATOR

(These two letters are considered lower-case by Unicode, but not by Java or SRFI 14.)

char-set:digitchar-set:digit

In Unicode, a character is a digit if it has the category Nd in the character attribute database. In Latin-1 and ASCII, the only such characters are 0123456789. In Unicode, there are other digit characters in other code blocks, such as Gujarati digits and Tibetan digits.

char-set:hex-digitchar-set:hex-digit

The only hex digits are 0123456789abcdefABCDEF.

char-set:letter+digitchar-set:letter+digit

The union of `char-set:letter` and `char-set:digit`.

char-set:graphicchar-set:graphic

A graphic character is one that would put ink on paper. The ASCII and Latin-1 graphic characters are the members of

```
char-set:letter
char-set:digit
char-set:punctuation
char-set:symbol
```

char-set:printingchar-set:printing

A printing character is one that would occupy space when printed, *i.e.*, a graphic character or a space character. `char-set:printing` is the union of `char-set:whitespace` and `char-set:graphic`.

char-set:whitespacechar-set:whitespace

In Unicode, a whitespace character is either

- a character with one of the space, line, or paragraph separator categories (Zs, Zl or Zp) of the Unicode character database.
- U+0009 Horizontal tabulation (\t control-I)
- U+000A Line feed (\n control-J)
- U+000B Vertical tabulation (\v control-K)
- U+000C Form feed (\f control-L)
- U+000D Carriage return (\r control-M)

There are 24 whitespace characters in Unicode 3.0:

0009	HORIZONTAL TABULATION	\t control-I
000A	LINE FEED	\n control-J
000B	VERTICAL TABULATION	\v control-K
000C	FORM FEED	\f control-L
000D	CARRIAGE RETURN	\r control-M
0020	SPACE	Zs
00A0	NO-BREAK SPACE	Zs
1680	OGHAM SPACE MARK	Zs
2000	EN QUAD	Zs
2001	EM QUAD	Zs
2002	EN SPACE	Zs
2003	EM SPACE	Zs
2004	THREE-PER-EM SPACE	Zs
2005	FOUR-PER-EM SPACE	Zs
2006	SIX-PER-EM SPACE	Zs
2007	FIGURE SPACE	Zs
2008	PUNCTUATION SPACE	Zs
2009	THIN SPACE	Zs
200A	HAIR SPACE	Zs
200B	ZERO WIDTH SPACE	Zs
2028	LINE SEPARATOR	Zl
2029	PARAGRAPH SEPARATOR	Zp
202F	NARROW NO-BREAK SPACE	Zs
3000	IDEOGRAPHIC SPACE	Zs

The ASCII whitespace characters are the first six characters in the above list – line feed, horizontal tabulation, vertical tabulation, form feed, carriage return, and space. These are also exactly the characters recognised by the Posix `isspace()` procedure. Latin-1 adds the no-break space.

Note: Java's `isWhitespace()` method is incompatible, including

0009	HORIZONTAL TABULATION	(\t control-I)
001C	FILE SEPARATOR	(control-\)
001D	GROUP SEPARATOR	(control-])
001E	RECORD SEPARATOR	(control-^)
001F	UNIT SEPARATOR	(control-_)

and excluding

00A0	NO-BREAK SPACE

Java's excluding the no-break space means that tokenizers can simply break character streams at "whitespace" boundaries. However, the exclusion introduces exceptions in other places, *e.g.* `char-set:printing` is no longer simply the union of `char-set:graphic` and `char-set:whitespace`.

char-set:iso-controlchar-set:iso-control

The ISO control characters are the Unicode/Latin-1 characters in the ranges [U+0000,U+001F] and [U+007F,U+009F].

ASCII restricts this set to the characters in the range [U+0000,U+001F] plus the character U+007F.

Note that Unicode defines other control characters which do not belong to this set (hence the qualifying prefix "iso-" in the name). This restriction is compatible with the Java `IsISOControl()` method.

char-set:punctuationchar-set:punctuation

In Unicode, a punctuation character is any character that has one of the punctuation categories in the Unicode character database (Pc, Pd, Ps, Pe, Pi, Pf, or Po.)

ASCII has 23 punctuation characters:

```
!"#%&'()*,-./:;?@[\]_{}
```

Latin-1 adds six more:

00A1	INVERTED EXCLAMATION MARK
00AB	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
00AD	SOFT HYPHEN
00B7	MIDDLE DOT
00BB	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
00BF	INVERTED QUESTION MARK

Note that the nine ASCII characters `$+<=>^`|~` are *not* punctuation. They are “symbols.”

char-set:symbolchar-set:symbol

In Unicode, a symbol is any character that has one of the symbol categories in the Unicode character database (Sm, Sc, Sk, or So). There are nine ASCII symbol characters:

```
$+<=>^`|~
```

Latin-1 adds 18 more:

00A2	CENT SIGN
00A3	POUND SIGN
00A4	CURRENCY SIGN
00A5	YEN SIGN
00A6	BROKEN BAR
00A7	SECTION SIGN
00A8	DIAERESIS
00A9	COPYRIGHT SIGN
00AC	NOT SIGN
00AE	REGISTERED SIGN
00AF	MACRON
00B0	DEGREE SIGN
00B1	PLUS-MINUS SIGN
00B4	ACUTE ACCENT
00B6	PILCROW SIGN
00B8	CEDILLA
00D7	MULTIPLICATION SIGN
00F7	DIVISION SIGN

char-set:blankchar-set:blank

Blank chars are horizontal whitespace. In Unicode, a blank character is either

- a character with the space separator category (Zs) in the Unicode character database.
- U+0009 Horizontal tabulation (\t control-I)

There are eighteen blank characters in Unicode 3.0:

0009	HORIZONTAL TABULATION	\t control-I
0020	SPACE	Zs
00A0	NO-BREAK SPACE	Zs
1680	OGHAM SPACE MARK	Zs
2000	EN QUAD	Zs
2001	EM QUAD	Zs
2002	EN SPACE	Zs
2003	EM SPACE	Zs
2004	THREE-PER-EM SPACE	Zs
2005	FOUR-PER-EM SPACE	Zs
2006	SIX-PER-EM SPACE	Zs
2007	FIGURE SPACE	Zs
2008	PUNCTUATION SPACE	Zs
2009	THIN SPACE	Zs
200A	HAIR SPACE	Zs
200B	ZERO WIDTH SPACE	Zs
202F	NARROW NO-BREAK SPACE	Zs
3000	IDEOGRAPHIC SPACE	Zs

The ASCII blank characters are the first two characters above – horizontal tab and space. Latin-1 adds the no-break space.

Java doesn't have the concept of "blank" characters, so there are no compatibility issues.

References & linksReferences & links

[**Java**] The following URLs provide documentation on relevant Java classes.
`http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Character.html`
`http://java.sun.com/products/jdk/1.2/docs/api/java/lang/String.html`
`http://java.sun.com/products/jdk/1.2/docs/api/java/lang/StringBuffer.html`
`http://java.sun.com/products/jdk/1.2/docs/api/java/text/Collator.html`
`http://java.sun.com/products/jdk/1.2/docs/api/java/text/package-summary.html`
[**MIT-Scheme**] `http://www.swiss.ai.mit.edu/projects/scheme/`
[**R5RS**] Revised^5 report on the algorithmic language Scheme.
R. Kelsey, W. Clinger, J. Rees (editors).
Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.
and ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998.
Available at `http://www.schemers.org/Documents/Standards/`.
[**SRFI**] The SRFI web site.
`http://srfi.schemers.org/`
[**SRFI-14**] SRFI-14: String libraries.
`http://srfi.schemers.org/srfi-14/`

This document, in HTML: `http://srfi.schemers.org/srfi-14/srfi-14.html`
This document, in plain text format: `http://srfi.schemers.org/srfi-14/srfi-14.txt`
Source code for the reference implementation: `http://srfi.schemers.org/srfi-14/srfi-14.scm`
Scheme 48 module specification, with typings: `http://srfi.schemers.org/srfi-14/srfi-14-s48-module.scm`
Regression-test suite: `http://srfi.schemers.org/srfi-14/srfi-14-tests.scm`

[**Unicode**] `http://www.unicode.org/`
[**UnicodeData**] The Unicode character database.
`ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.txt`
`ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.html`

Hash tables

[**Red Edition item 7**] What hash table library should R7RS-large provide?

> SRFI 69 is a basic hash table library. SRFI 125 is an upward compatible extension of it. Likewise, R6RS provides a basic hash table library, of which SRFI 126 is an upward compatible extension. Since the interface of SRFI 125 depends on SRFI 128, voting for SRFI 125 will cause your vote on comparator libraries to be automatically changed to SRFI 128.

TODO This section specifies some procedures that are then labelled "deprecated". Should they remain?

TODO This section makes references to R6RS and Common Lisp. Should those references remain?

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

- Is disjoint from all other types.
- Provides a mapping from objects known as *keys* to corresponding objects known as *values*.
  - Keys may be any Scheme objects in some kinds of hash tables, but are restricted in other kinds.
  - Values may be any Scheme objects.
- Has no intrinsic order for the key-value *associations* it contains.
- Provides an *equality predicate* which defines when a proposed key is the same as an existing key. No table may contain more than one value for a given key.
- Provides a *hash function* which maps a candidate key into a non-negative exact integer.
- Supports mutation as the primary means of setting the contents of a table.
- Provides key lookup and destructive update in (expected) amortized constant time, provided a satisfactory hash function is available.
- Does not guarantee that whole-table operations work in the presence of concurrent mutation of the whole hash table (values may be safely mutated).

The names described in this section comprise the `(scheme hash-table)` library.

Pronunciation

The slash in the names of some procedures can be pronounced "with".

Specification

The procedures in this SRFI are in the `(srfi 125)` library (or `(srfi :125)` on R6RS).

All references to "executing in expected amortized constant time" presuppose that a satisfactory hash function is available. Arbitrary or impure hash functions can make a hash of any implementation.

Hash tables are allowed to cache the results of calling the equality predicate and hash function, so programs cannot rely on the hash function being called exactly once for every primitive hash table operation: it may be called zero, one, or more times.

It is an error if the procedure argument of `hash-table-find`, `hash-table-count`, `hash-table-map`, `hash-table-for-each`, `hash-table-map!`, `hash-table-map->list`, `hash-table-fold` or `hash-table-prune!` mutates the hash table being walked.

It is an error to pass two hash tables that have different comparators or equality predicates to any of the procedures of this SRFI.

Implementations are permitted to ignore user-specified hash functions in certain circumstances. Specifically, if the equality predicate, whether passed as part of a comparator or explicitly, is more fine-grained (in the sense of R7RS-small section 6.1) than `equal?`, the implementation is free — indeed, is encouraged — to ignore the user-specified hash function and use something implementation-dependent. This allows the use of addresses as hashes, in which case the keys must be rehashed if they are moved by the garbage collector. Such a hash function is unsafe to use outside the context of implementation-provided hash tables. It can of course be exposed by an implementation as an extension, with suitable warnings against inappropriate uses.

It is an error to mutate a key during or after its insertion into a hash table in such a way that the hash function of the table will return a different result when applied to that key.

Index

- Constructors: `make-hash-table`, `hash-table`, `hash-table-unfold`, `alist->hash-table`
- Predicates: `hash-table?`, `hash-table-contains?`, `hash-table-exists?` (deprecated), `hash-table-empty?`, `hash-table=?`, `hash-table-mutable?`
- Accessors: `hash-table-ref`, `hash-table-ref/default`
- Mutators: `hash-table-set!`, `hash-table-delete!`, `hash-table-intern!`, `hash-table-update!`, `hash-table-update!/defau` `hash-table-pop!`, `hash-table-clear!`
- The whole hash table: `hash-table-size`, `hash-table-keys`, `hash-table-values`, `hash-table-entries`, `hash-table-find`, `hash-table-count`
- Mapping and folding: `hash-table-map`, `hash-table-for-each`, `hash-table-walk` (deprecated), `hash-table-map!`, `hash-table-map->list`, `hash-table-fold`, `hash-table-prune!`
- Copying and conversion: `hash-table-copy`, `hash-table-empty-copy`, `hash-table->alist`
- Hash tables as sets: `hash-table-union!`, `hash-table-merge!` (deprecated), `hash-table-intersection!`, `hash-table-differ` `hash-table-xor!`
- Hash functions and reflectivity (deprecated): `hash`, `string-hash`, `string-ci-hash`, `hash-by-identity`, `hash-table-equivalen` `hash-table-hash-function`

Constructors

(`make-hash-table` *comparator arg …*) → *hash-table???*	hash-table library procedure
(`make-hash-table` *comparator*) → *hash-table???*	hash-table library procedure
(`make-hash-table` *equality-predicate hash-function arg …*) → *hash-table???*	hash-table library procedure
(`make-hash-table` *equality-predicate hash-function*) → *hash-table???*	hash-table library procedure
(`make-hash-table` *equality-predicate arg …*) → *hash-table???*	hash-table library procedure
(`make-hash-table` *equality-predicate*) → *hash-table???*	hash-table library procedure

Returns a newly allocated hash table whose equality predicate and hash function are extracted from *comparator*. Alternatively, for backward compatibility with SRFI 69 the equality predicate and hash function can be passed as separate arguments; this usage is deprecated.

As mentioned above, implementations are free to use an appropriate implementation-dependent hash function instead of the specified hash function, provided that the specified equality predicate is a refinement of the `equal?` predicate. This applies whether the hash function and equality predicate are passed as separate arguments or packaged up into a comparator.

If an equality predicate rather than a comparator is provided, the ability to omit the *hash-function* argument is severely limited. The implementation must provide hash functions appropriate for use with the predicates `eq?`, `eqv?`, `equal?`, `string=?`, and `string-ci=?`, and may extend this list. But if any unknown equality predicate is provided without a hash function, an error should be signaled. The constraints on equality predicates and hash functions are given in SRFI 128.

The meaning of any further arguments is implementation-dependent. However, implementations which support the ability to specify the initial capacity of a hash table should interpret a non-negative exact integer as the specification of that capacity. In addition, if the symbols `thread-safe`, `weak-keys`, `ephemeral-keys`, `weak-values`, or `ephemeral-values` are present, implementations should create thread-safe hash tables, hash tables with weak keys or ephemeral keys, or hash tables with weak or ephemeral values respectively. Implementations are free to use ephemeral keys or values when weak keys or values respectively have been requested. To avoid collision with the *hash-function* argument, none of these arguments can be procedures.

(R6RS `make-eq-hashtable`, `make-eqv-hashtable`, and `make-hashtable`; Common Lisp `make-hash-table`)

(`hash-table` *comparator key value …*) → *hash-table???* hash-table library procedure

TODO I've been converting entry descriptions to match R7RS, with separate prototypes for omitted arguments, that doesn't work here, because the number of keys and values doesn't match. Fix somehow.

Returns a newly allocated hash table, created as if by `make-hash-table` using *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 implementation supports immutable hash tables, this procedure returns an immutable hash table. 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 arg …*) → *???* hash-table library procedure

TODO This one looks garbled from the Pandoc conversion. Check against SRFI.

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 …*) → *hash-table???* hash-table library procedure
(`alist->hash-table` *alist equality-predicate hash-function arg …*) → *hash-table???* hash-table library procedure
(`alist->hash-table` *alist equality-predicate hash-function*) → *hash-table???* hash-table library procedure
(`alist->hash-table` *alist equality-predicate arg …*) → *hash-table???* hash-table library procedure
(`alist->hash-table` *alist equality-predicate*) → *hash-table???* hash-table library procedure

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. The second form TODO and subsequent forms is for compatibility with SRFI 69, and is deprecated.

Predicates

(`hash-table?` *obj*) → *boolean???* hash-table library procedure

Returns `#t` if *obj* is a hash table, and `#f` otherwise. (R6RS `hashtable?`; Common Lisp `hash-table-p`)

(`hash-table-contains?` *hash-table key*) → *boolean???* hash-table library procedure
(`hash-table-exists?` *hash-table key*) → *boolean???* hash-table library procedure

Returns `#t` if there is any association to *key* in *hash-table*, and `#f` otherwise. Must execute in expected amortized constant time. The `hash-table-exists?` procedure is the same as `hash-table-contains?`, is provided for backward compatibility with SRFI 69, and is deprecated. (R6RS `hashtable-contains?`)

(`hash-table-empty?` *hash-table*) → *boolean???* hash-table library procedure

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

`(hash-table=?` *value-comparator hash-table*$_1$ *hash-table*$_2$`)` → *boolean???* hash-table library procedure

Returns `#t` if *hash-table*$_1$ and *hash-table*$_2$ 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*`)` → *boolean???* hash-table library procedure

Returns `#t` if the hash table is mutable. Implementations may or may not support immutable hash tables. (R6RS `hashtable-mutable?`)

Accessors

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

`(hash-table-ref` *hash-table key failure success*`)` → *value???* hash-table library procedure
`(hash-table-ref` *hash-table key failure*`)` → *value???* hash-table library procedure
`(hash-table-ref` *hash-table key*`)` → *value???* hash-table library procedure

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. Otherwise, it is an error. Must execute in expected amortized constant time, not counting the time to call the procedures. SRFI 69 does not support the *success* procedure.

`(hash-table-ref/default` *hash-table key default*`)` → *value???* hash-table library procedure

Semantically equivalent to, but may be more efficient than, the following code:

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

(R6RS `hashtable-ref`; Common Lisp `gethash`)

Mutators

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

`(hash-table-set!` *hash-table arg …*`)` → *unspecified* hash-table library procedure

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. Must execute in expected amortized constant time per key. SRFI 69, R6RS `hashtable-set!` and Common Lisp `(setf gethash)` do not handle multiple associations.

`(hash-table-delete!` *hash-table key …*`)` → *integer* hash-table library procedure

Deletes any association to each *key* in *hash-table* and returns the number of keys that had associations. Must execute in expected amortized constant time per key. SRFI 69, R6RS `hashtable-delete!`, and Common Lisp `remhash` do not handle multiple associations.

`(hash-table-intern!` *hash-table key failure*`)` → *value???* hash-table library procedure

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*. Must execute in expected amortized constant time.

TODO If failure is invoked, is that the return value of hash-table-intern!?

`(hash-table-update!` *hash-table key updater failure success*`)` → *unspecified???* hash-table library procedure
`(hash-table-update!` *hash-table key updater failure*`)` → *unspecified???* hash-table library procedure
`(hash-table-update!` *hash-table key updater*`)` → *unspecified???* hash-table library procedure

Semantically equivalent to, but may be more efficient than, the following code:

```
(hash-table-set! hash-table key (updater (hash-table-ref hash-table key failure success)))
```

Must execute in expected amortized constant time. Returns an unspecified value. (SRFI 69 and R6RS `hashtable-update!` do not support the *success* procedure)

`(hash-table-update!/default` *hash-table key updater default*`)` → *unspecified???* hash-table library procedure

Semantically equivalent to, but may be more efficient than, the following code:

```
(hash-table-set! hash-table key (updater (hash-table-ref/default hash-table key default)))
```

Must execute in expected amortized constant time. Returns an unspecified value.

`(hash-table-pop!` *hash-table*`)` → *key??? value???* hash-table library procedure

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

`(hash-table-clear!` *hash-table*`)` → *unspecified???* hash-table library procedure

Delete all the associations from *hash-table*. (R6RS `hashtable-clear!`; Common Lisp `clrhash`)

The whole hash table

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

`(hash-table-size` *hash-table*`)` → *integer???* hash-table library procedure

Returns the number of associations in *hash-table* as an exact integer. Should execute in constant time. (R6RS `hashtable-size`; Common Lisp `hash-table-count`.)

`(hash-table-keys` *hash-table*`)` → *list* hash-table library procedure

Returns a newly allocated list of all the keys in *hash-table*. R6RS `hashtable-keys` returns a vector.

`(hash-table-values` *hash-table*`)` → *list* hash-table library procedure

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

`(hash-table-entries` *hash-table*`)` → *list list* hash-table library procedure

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. R6RS `hash-table-entries` returns vectors.

`(hash-table-find` *proc hash-table failure*`)` → *value or* `#f` *???* hash-table library procedure

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*`)` → *integer???* hash-table library procedure

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

Mapping and folding

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

(`hash-table-map` *proc comparator hash-table*) → *hash-table???* hash-table library procedure

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.

(`hash-table-for-each` *proc hash-table*) → *unspecified???* hash-table library procedure
(`hash-table-walk` *hash-table proc*) → *unspecified* hash-table library procedure

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. The `hash-table-walk` procedure is equivalent to `hash-table-for-each` with the arguments reversed, is provided for backward compatibility with SRFI 69, and is deprecated. (Common Lisp `maphash`)

(`hash-table-map!` *proc hash-table*) → *unspecified???* hash-table library procedure

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*) → *list* hash-table library procedure

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*) → *value???* hash-table library procedure
(`hash-table-fold` *hash-table proc seed*) → *value???* hash-table library procedure

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*. The order of arguments with *hash-table* as the first argument is provided for SRFI 69 compatibility, and is deprecated.

(`hash-table-prune!` *proc hash-table*) → *unspecified???* hash-table library procedure

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

Copying and conversion

(`hash-table-copy` *hash-table mutable?*) → *hash-table???* hash-table library procedure
(`hash-table-copy` *hash-table?*) → *hash-table???* hash-table library procedure

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. SRFI 69 `hash-table-copy` does not support a second argument. (R6RS `hashtable-copy`)

(`hash-table-empty-copy` *hash-table*) → *hash-table???* hash-table library procedure

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

(`hash-table->alist` *hash-table*) → *alist???* hash-table library procedure

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

Hash tables as sets

`(hash-table-union!` *hash-table*$_1$ *hash-table*$_2$`)` → *hash-table???* hash-table library procedure
`(hash-table-merge!` *hash-table*$_1$ *hash-table*$_2$`)` → *hash-table???* hash-table library procedure

Adds the associations of *hash-table*$_2$ to *hash-table*$_1$ and returns *hash-table*$_1$. If a key appears in both hash tables, its value is set to the value appearing in *hash-table*$_1$. Returns *hash-table*$_1$. The `hash-table-merge!` procedure is the same as `hash-table-union!`, is provided for compatibility with SRFI 69, and is deprecated.

`(hash-table-intersection!` *hash-table*$_1$ *hash-table*$_2$`)` → *hash-table???* hash-table library procedure

Deletes the associations from *hash-table*$_1$ which don't also appear in *hash-table*$_2$ and returns *hash-table*$_1$.

`(hash-table-difference!` *hash-table*$_1$ *hash-table*$_2$`)` → *hash-table* hash-table library procedure

Deletes the associations of *hash-table*$_1$ whose keys are also present in *hash-table*$_2$ and returns *hash-table*$_1$.

`(hash-table-xor!` *hash-table*$_1$ *hash-table*$_2$`)` → *hash-table* hash-table library procedure

Deletes the associations of *hash-table*$_1$ whose keys are also present in *hash-table*$_2$, and then adds the associations of *hash-table*$_2$ whose keys are not present in *hash-table*$_1$ to *hash-table*$_1$. Returns *hash-table*$_1$.

Hash functions and reflectivity

TODO If we are removing deprecated features, then this whole section goes. If it remains, I will reformat it.

These functions are made part of this SRFI solely for compatibility with SRFI 69, and are deprecated.

`(hash` *obj*`)`

The same as SRFI 128's `default-hash` procedure, except that it must accept (and should ignore) an optional second argument.

`(string-hash` *obj*`)`

Similar to SRFI 128's `string-hash` procedure, except that it must accept (and should ignore) an optional second argument. It is incompatible with the procedure of the same name exported by SRFI 128 and SRFI 126.

`(string-ci-hash` *obj*`)`

Similar to SRFI 128's `string-ci-hash` procedure, except that it must accept (and should ignore) an optional second argument. It is incompatible with the procedure of the same name exported by SRFI 128 and SRFI 126.

`(hash-by-identity` *obj*`)`

The same as SRFI 128's `default-hash` procedure, except that it must accept (and should ignore) an optional second argument. However, if the implementation replaces the hash function associated with the `eq?` predicate with an implementation-dependent alternative, it is an error to call this procedure at all.

`(hash-table-equivalence-function` *hash-table*`)`

Returns the equivalence procedure used to create *hash-table*.

`(hash-table-hash-function` *hash-table*`)`

Returns the hash function used to create *hash-table*. However, if the implementation has replaced the user-specified hash function with an implementation-specific alternative, the implementation may return `#f` instead.

Editor: Arthur A. Gleckler

5. Immutability

Immutable pairs and lists

[**Red Edition item 8**] What immutable pair/list library should R7RS-large provide?

SRFI 116 is a variant of SRFI 1 for immutable pairs, which are a separate data type from ordinary Scheme pairs. Only a few functional update procedures are provided. Procedure names are prefixed with "i".

TODO Much of this SRFI repeats text from SRFI-1. Could some of it be reduced?

Post-finalization note: One sentence has been added to the section "The procedures" below, and several errors in the examples have been silently corrected. These do not affect the conformance conditions for this SRFI.

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.

The names described in this section comprise the `(scheme ilist)` library.

Procedure Index

Here is a short list of the procedures provided by this SRFI.

Constructors `ipair ilist`
```
xipair ipair* make-ilist ilist-tabulate
ilist-copy iiota
```

Predicates `ipair?`
```
proper-ilist?/ilist? dotted-ilist?
not-ipair? null-ilist?
ilist=
```

Selectors `icar icdr ... icddadr icddddr ilist-ref`
```
ifirst isecond ithird ifourth ififth
isixth iseventh ieighth ininth itenth
icar+icdr
itake       idrop/ilist-tail
itake-right idrop-right
isplit-at
ilast last-ipair
```

Miscellaneous: length, append, concatenate, reverse, zip & count `ilength`
```
iappend  iconcatenate  ireverse  iappend-reverse
izip iunzip1 iunzip2 iunzip3 iunzip4 iunzip5
icount
```

Fold, unfold & map `imap         ifor-each`
```
ifold       iunfold        ipair-fold       ireduce
ifold-right iunfold-right  ipair-fold-right ireduce-right
iappend-map ipair-for-each ifilter-map      imap-in-order
```

Filtering & partitioning `ifilter  ipartition  iremove`

Searching `imember imemq imemv`
```
ifind         ifind-tail
iany ievery
ilist-index
itake-while   idrop-while
ispan ibreak
```

Deleting `idelete       idelete-duplicates`

Immutable association lists `iassoc iassq iassv`
`ialist-cons` `ialist-delete`

Replacement `replace-icar replace-icdr`

Conversion `pair->ipair` `ipair->pair`
`list->ilist` `ilist->list`
`tree->itree` `itree->tree`
`gtree->itree gtree->tree`

Procedure application `iapply`

Comparators `ipair-comparator` `ilist-comparator`
`make-ilist-comparator` `make-improper-ilist-comparator`
`make-icar-comparator` `make-icdr-comparator`

Quotation

TODO Sort out where this goes. Some of this definitely belongs in the Rationale, while some of the rest belongs here. Rather than "recommending", something slightly stronger is needed?

The various Scheme standards permit, but do not require, Scheme implementations to treat quoted pairs and lists as immutable. Thus whereas the expression `(let ((foo (list 1 2 3))) (set-car! foo 10) foo)` evaluates to `(10 2 3)`, the value of `(let ((foo '(1 2 3))) (set-car! foo 10) foo)` is not portable, and is in fact an error.

This SRFI recommends that implementations that provide both this SRFI and immutable quotations should cause quotations to return the same immutable pairs that this SRFI describes. This means that the standard Scheme pair and list operations, as well as libraries like SRFI 1 which are built on them, should accept both mutable and immutable pairs: thus `(car (ilist 1 2))` should evaluate to `1`.

This SRFI further recommends that `read` should return mutable pairs and lists when reading list structure. No recommendation is made about the behavior of `write`, `display`, and similar output procedures on immutable lists.

To make life easier for Scheme programmers, given that many implementations do not provide immutable quotation, the syntax keyword `iq` is provided as part of this SRFI. It is analogous to `quote`, taking an arbitrary number of literals and constructing an ilist from them, with any pairs in the literals converted to ipairs. It is useful for providing constant ipair-based objects. Note that pairs within literal vectors or other implementation-dependent literals will not be converted. Unfortunately, there is no ilist analogue of `'`, so we save keystrokes by using `iq` rather than `iquote` and omitting the top-level parentheses.

Notation

The templates given below obey the following conventions for procedure formals:

ilist	A proper (`()`-terminated) ilist
dilist	A proper or dotted ilist
ipair	An immutable pair
x, y, d, a	Any value
object, value	Any value
n, i	A natural number (an integer >= 0)
proc	A procedure
pred	A procedure whose return value is treated as a boolean
=	A boolean procedure taking two arguments

To interpret the examples, pretend that they are executed on a Scheme that prints immutable pairs and lists with the syntax of mutable ones.

It is an error to pass a dotted ilist to a procedure not defined to accept such an argument.

Added after finalization: Implementers should extend the Scheme predicate `equal?` to descend into immutable pairs in the same way that it descends into mutable pairs.

Constructors

`(ipair` *a d*`)` → *ipair* ilist library procedure

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 …*`)` → *ilist* ilist library procedure

Returns a newly allocated ilist of its arguments.

```
(ilist 'a (+ 3 4) 'c) =>  (a 7 c)
(ilist)               =>  ()
```

`(xipair` *d a*`)` → *ipair* ilist library procedure

```
(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*` *elt*$_1$ *elt*$_2$ *…*`)` → *object* ilist library procedure

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*`)` → *ilist* ilist library procedure
`(make-ilist` *n*`)` → *ilist* ilist library procedure

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*`)` → *ilist* ilist library procedure

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*`)` → *dilist* ilist library procedure

Copies the spine of the argument, including the ilist tail.

`(iiota` *count start step*`)` → *ilist* ilist library procedure
`(iiota` *count start*`)` → *ilist* ilist library procedure
`(iiota` *count*`)` → *ilist* ilist library procedure

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)
```

Predicates

`(proper-ilist?` *x*`)` → *boolean* ilist library procedure
`(ilist?` *x*`)` → *boolean* ilist library procedure

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*`)` → *boolean* ilist library procedure

True if x is a finite, non-nil-terminated ilist. That is, there exists an n >= 0 such that icdr^n(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*`)` → *boolean* ilist library procedure

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*`)` → *boolean* ilist library procedure

Ilist is a proper ilist. This procedure returns true if the argument is the empty list (), and `#f` 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*`)` → *boolean* ilist library procedure

```
(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.

`(e`*l*`)` → = ilist library t
ilist$_1$ ...proceduren

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 ilist$_i$, then its second argument is from ilist$_{i+1}$, *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 ilist$_i$ is compared to ilist$_{i+1}$ (as opposed, for example, to comparing ilist$_1$ to ilist$_i$, 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
```

Selectors

`(icar` *ipair*`)` → *value* ilist library procedure
`(icdr` *ipair*`)` → *value* ilist library procedure

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*`)` → *value* ilist library procedure
`(icadr` *ipair*`)` → *value* ilist library procedure
:
`(icdddar` *ipair*`)` → *value* ilist library procedure
`(icddddr` *ipair*`)` → *value* ilist library procedure

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*`)` → *value* ilist library procedure

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*) → *object*	ilist library procedure
(`isecond`*ipair*) → *object*	ilist library procedure
(`ithird`*ipair*) → *object*	ilist library procedure
(`ifourth`*ipair*) → *object*	ilist library procedure
(`ififth`*ipair*) → *object*	ilist library procedure
(`isixth`*ipair*) → *object*	ilist library procedure
(`iseventh`*ipair*) → *object*	ilist library procedure
(`ieighth`*ipair*) → *object*	ilist library procedure
(`ininth`*ipair*) → *object*	ilist library procedure
(`itenth`*ipair*) → *object*	ilist library procedure

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

```
(ithird '(a b c d e)) => c
```

(`icar+icdr` *ipair*) → *x y* ilist library procedure

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*) → *ilist*	ilist library procedure
(`idrop` *x i*) → *object*	ilist library procedure
(`ilist-tail` *x i*) → *object*	ilist library procedure

`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.

```
(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:

```
(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*) → *object*	ilist library procedure
(`idrop-right` *dilist i*) → *ilist*	ilist library procedure

`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*) → *ilist object* ilist library procedure

`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*) → *object* ilist library procedure
(`last-ipair` *ipair*) → *ipair* ilist library procedure

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)
```

Miscellaneous: length, append, concatenate, reverse, zip & count

(`ilength`*ilist*) → *integer* ilist library procedure

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` *ilist*$_1$ *...*) → *ilist* ilist library procedure

Returns an ilist consisting of the elements of ilist$_1$ 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 ilist$_i$ 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*) → *value* ilist library procedure

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*`)` → *ilist* ilist library procedure

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*`)` → *ilist* ilist library procedure

`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` *ilist*$_1$ *ilist*$_2$ *...*`)` → *ilist* ilist library procedure

```
(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*`)` → *ilist* ilist library procedure
`(iunzip2` *ilist*`)` → *ilist ilist* ilist library procedure
`(iunzip3` *ilist*`)` → *ilist ilist ilist* ilist library procedure
`(iunzip4` *ilist*`)` → *ilist ilist ilist ilist* ilist library procedure
`(iunzip5` *ilist*`)` → *ilist ilist ilist ilistilist* ilist library procedure

`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 ilist*$_1$ *ilist*$_2$ *...*`)` → *integer* ilist library procedure

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
```

Fold, unfold, and map

`(ifold` *kons knil ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

The fundamental ilist iterator.

First, consider the single ilist-parameter case. If ilist$_1$ = (e_1 e_2 ...e_n), 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 ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

The fundamental ilist recursion operator.

First, consider the single ilist-parameter case. If ilist$_1$ = `(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 ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

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 ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

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*`)` → *value* ilist library procedure

`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*`)` → *value* ilist library procedure

`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]*`)` → *ilist* ilist library procedure

`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), g^2(seed), g^3(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*`)` → *ilist* ilist library procedure
`(iunfold-right` *p f g seed*`)` → *ilist* ilist library procedure

`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), g^2(seed), g^3(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* *ilist*$_1$ *ilist*$_2$ ...) → *ilist* ilist library procedure

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 ilist*$_1$ *ilist*$_2$ *...*`)` → *unspecified* ilist library procedure

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 ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

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 ilist*$_1$ *ilist*$_2$ *...*`)` → *ilist* ilist library procedure

A variant of the `imap` procedure that guarantees to apply f across the elements of the ilist$_i$ 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 ilist*$_1$ *ilist*$_2$ *...*`)` → *unspecific* ilist library procedure

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 ilist*$_1$ *ilist*$_2$ *...*`)` → *ilist* ilist library procedure

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.

Filtering and partitioning

`(ifilter` *pred ilist*`)` → *ilist* ilist library procedure

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*`)` → *ilist ilist* ilist library procedure

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*`)` → *ilist* ilist library procedure

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)
```

Searching

The following procedures all search ilists for a leftmost element satisfying some criteria. This means they do not always examine the entire ilist; thus, there is no efficient way for them to reliably detect and signal an error when passed a dotted ilist. Here are the general rules describing how these procedures work when applied to different kinds of ilists:

Proper ilists: The standard, canonical behavior happens in this case.

Dotted ilists: It is an error to pass these procedures a dotted ilist that does not contain an element satisfying the search criteria. That is, it is an error if the procedure has to search all the way to the end of the dotted ilist. However, this SRFI does *not* specify anything at all about the behavior of these procedures when passed a dotted ilist containing an element satisfying the search criteria. It may finish successfully, signal an error, or perform some third action. Different implementations may provide different functionality in this case; code which is compliant with this SRFI may not rely on any particular behavior. Future SRFIs may refine this SRFI to define specific behavior in this case.

In brief, compliant code may not pass a dotted ilist argument to these procedures.

Here are some examples, using the `ifind` and `iany` procedures as canonical representatives:

```
;; Proper ilist - success
(ifind even? (iq 1 2 3))    => 2
(iany  even? (iq 1 2 3))    => #t

;; proper ilist - failure
(ifind even? (iq 1 7 3))    => #f
(iany  even? (iq 1 7 3))    => #f

;; Failure is error on a dotted ilist.
(ifind even? (ipair 1 (ipair 3 'x)))    => error
(iany  even? (ipair 1 (ipair 3 'x)))    => error

;; The dotted ilist contains an element satisfying the search.
;; This case is not specified - it could be success, an error,
;; or some third possibility.
(ifind even? (ipair 1 (ipair 2 'x)))    => error/undefined
(iany  even? (ipair 1 (ipair 2 'x)))    => error/undefined ; success, error or other.
```

(`ifind` *pred ilist*) → *value* ilist library procedure

Return the first element of ilist that satisfies predicate pred; `#f` 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*) → *ipair or* `#f` ilist library procedure

Return the first ipair of ilist whose icar satisfies pred. If no ipair does, return `#f`.

`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)
```

`Ifind-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*) → *ilist* ilist library procedure

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*) → *ilist* ilist library procedure

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*`)` → *ilist ilist* ilist library procedure
`(ibreak` *pred ilist*`)` → *ilist ilist* ilist library procedure

`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 ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

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

If there are n ilist arguments ilist$_1$ ... ilist$_n$, then pred must be a procedure taking n arguments and returning a boolean result.

`iany` applies pred to the first elements of the ilist$_i$ parameters. If this application returns a true value, `iany` immediately returns that value. Otherwise, it iterates, applying pred to the second elements of the ilist$_i$ 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 ilist*$_1$ *ilist*$_2$ *...*`)` → *value* ilist library procedure

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

If there are n ilist arguments ilist$_1$ ...ilist$_n$, then pred must be a procedure taking n arguments and returning a boolean result.

`ievery` applies pred to the first elements of the ilist$_i$ parameters. If this application returns `#f`, `ievery` immediately returns `#f`. Otherwise, it iterates, applying pred to the second elements of the ilist$_i$ parameters, then the third, and so forth. The iteration stops when a `#f` 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 ilist$_i$ 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* *ilist*$_1$ *ilist*$_2$ *...*) → *integer or* `#f` ilist library procedure

Return the index of the leftmost element that satisfies pred.

If there are n ilist arguments ilist$_1$ ...ilist$_n$, then pred must be a function taking n arguments and returning a boolean result.

`ilist-index` applies pred to the first elements of the ilist$_i$ parameters. If this application returns true, `ilist-index` immediately returns zero. Otherwise, it iterates, applying pred to the second elements of the ilist$_i$ 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* *=*) → *ilist* ilist library procedure
(`imember` *x* *ilist*) → *ilist* ilist library procedure
(`imemq` *x* *ilist*) → *ilist* ilist library procedure
(`imemv` *x* *ilist*) → *ilist* ilist library procedure

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 e$_i$ 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.
```

Deletion

(`idelete`*x* *ilist* *[=]*) → *ilist* ilist library procedure

`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 e_i is not specified. Thus, one can reliably remove all the numbers greater than five from an ilist with `(idelete 5 ilist <)`

`(idelete-duplicates`*ilist =*`)` → *ilist* ilist library procedure
`(idelete-duplicates`*ilist*`)` → *ilist* ilist library procedure

`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 $O(n^2)$ 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))
```

Immutable association lists

An "immutable association list" (or "ialist") is an ilist of ipairs. The icar of each ipair contains a key value, and the icdr contains the associated data value. They can be used to construct simple look-up tables in Scheme. Note that ialists are probably inappropriate for performance-critical use on large data; in these cases, immutable maps or some other alternative should be employed.

`(iassoc` *key ialist =*`)` → *ipair or* `#f` ilist library procedure
`(iassoc` *key ialist*`)` → *ipair or* `#f` ilist library procedure
`(iassq` *key ialist*`)` → *ipair or* `#f` ilist library procedure
`(iassv` *key ialist*`)` → *ipair or* `#f` ilist library procedure

ialist must be an immutable association list—an ilist of ipairs. These procedures find the first ipair in ialist whose icar field is key, and returns that ipair. If no ipair in ialist has key as its icar, then `#f` is returned. `iassq` uses `eq?` to compare key with the icar fields of the ipairs in ialist, while `iassv` uses `eqv?` and `iassoc` uses `equal?`.

```
(define e (iq (a 1) (b 2) (c 3)))
(iassq 'a e)                              =>  (a 1)
(iassq 'b e)                              =>  (b 2)
(iassq 'd e)                              =>  #f
(iassq (ilist 'a) (iq ((a)) ((b)) ((c)))  =>  #f
(iassoc '(a) (ilist '((a)) '((b)) '((c)))) =>  ((a))
(iassq 5 (iq (2 3) (5 7) (11 13)))     =>  *unspecified*
(iassv 5 (iq (2 3) (5 7) (11 13)))     =>  (5 7)
```

The comparison procedure is used to compare the elements e_i of ilist to the key parameter in this way:

```
 (= key (icar ei))  ; ilist is (E1 ... En)
```

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

Note that fully general ialist searching may be performed with the `ifind-tail` and `ifind` procedures, *e.g.*

```
;; Look up the first association in ialist with an even key:
(ifind (lambda (a) (even? (icar a))) ialist)
```

`(ialist-cons` *key datum ialist*`)` → *ialist* ilist library procedure

```
(lambda (key datum ialist) (ipair (ipair key datum) ialist))
```

Construct a new ialist entry mapping key -> datum onto ialist. `(ialist-delete`*key ialist =*`)` → *ialist*

ilist library procedure
`(ialist-delete`*key ialist*`)` → *ialist* ilist library procedure

`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 k_i 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 <)`

Replacement

These two procedures are analogues of the primitive side-effect operations on pairs, `set-car!` and `set-cdr!`.

`(replace-icar` *ipair object*`)` → *ipair* ilist library procedure

This procedure returns an ipair with object in the icar field and the icdr of ipair in the icdr field.

`(replace-icdr` *ipair object*`)` → *ipair* ilist library procedure

This procedure returns an ipair with object in the icdr field and the icar of ipair in the icar field.

Conversion

These procedures convert between mutable and immutable pair structures.

`(pair->ipair` *pair*`)` → *ipair* ilist library procedure
`(ipair->pair` *ipair*`)` → *pair* ilist library procedure

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*`)` → *dilist* ilist library procedure
`(ilist->list` *dilist*`)` → *flist* ilist library procedure

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*`)` → *object* — ilist library procedure
`(itree->tree` *object*`)` → *object* — ilist library procedure

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*`)` → *object* — ilist library procedure
`(gtree->tree` *object*`)` → *object* — ilist library procedure

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.

Procedure Application

This procedure allows a procedure to be applied to an ilist.

`(iapply` *procedure object …ilist*`)` → *object* — ilist library procedure

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.

Comparators

`ipair-comparator` — ilistobject

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` — ilistobject

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*`)` → *comparator* — ilist library procedure

The `make-ilist-comparator` procedure returns a comparator suitable for comparing ilists using element-comparator to compare the elements.

`(make-improper-ilist-comparator` *comparator*`)` → *comparator* ilist library procedure

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*`)` → *comparator* ilist library procedure

The `make-icar-comparator` procedure returns a comparator that compares ipairs on their icars alone using *comparator*.

`(make-icdr-comparator` *comparator*`)` → *comparator* ilist library procedure

The `make-icdr-comparator` procedure returns a comparator that compares ipairs on their icdrs alone using *comparator*.

References & links

This document, in HTML: http://srfi.schemers.org/srfi-116/srfi-116.html
Source code for the reference implementation: `http://srfi.schemers.org/srfi-116/ilists.tar.gz`
Archive of SRFI-116 discussion-list email: `http://srfi.schemers.org/srfi-116/mail-archive/maillist.html`
SRFI web site: `http://srfi.schemers.org/`

Editor: Mike Sperber

Random access pairs and lists

[**Red Edition item 9**] What random access pair/list library should R7RS-large provide?

> SRFI 101 is a library for purely functional pairs and lists with O(log n) access time. To avoid conflicts, procedure names will be modified for R7RS-large purposes as follows:

- `make-list` to become `make-rlist`
- `random-access-list->linear-access-list` to become `rlist->list`
- `linear-access-list->random-access-list` to become `list->rlist`
- all other identifiers to be prefixed with `r` (`rcons`, `rpair?`, `rcar`, `rmap`, etc.)

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 R^{6}RS [2] base library and list utility standard library [3].

The names described in this section comprise the `(scheme rlist)` library.

TODO The name changes given above have not yet been incorporated into this document.

TODO This library somehow redefines features found in R^{7}RS small. I'm not sure how to describe that here. I'm leaving this SRFI for the next iteration.

Random-access pairs and lists

A *random-access pair* (or just *pair*) is a compound structure with two fields called the car and the cdr fields (consistent with the historical naming of pair fields in Scheme). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`.

Pairs are used primarily to represents lists. A list can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set *X* such that

- The empty list is in *X*.
- If *list* is in *X*, then any pair whose cdr field contains *list* is also in *X*.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type. It is not a pair. It has no elements and its length is zero.

> *Note:* The above definitions imply that all lists have finite length and are terminated by the empty list.

A chain of pairs is defined recursively as either a non-pair object or a pair whose cdr is a chain of pairs (Note: *every value* is a chain of pairs). A chain of pairs ending in the empty list is a list. A chain of pairs not ending in the empty list is called an improper list. Note that an improper list is not a list. Whether a given pair is a list depends upon what is stored in the cdr field.

The external representation of pairs is not specified by this SRFI, however the examples below do use the typical notation for writing pair and list values.

Random-access pairs and lists are specified to be fully functional, or, to use the term from the academic literature, *fully persistent* [1]. Full persistence means that all operations on random-access lists, notably including `cons`, `list-ref`, `list-set`, and `list-ref/update`, are specified

1. not to mutate any of their arguments; perforce
2. to be safe to execute concurrently on shared arguments; and
3. to suffer no degradation of performance as a consequence of the history of operations carried out to produce their arguments (except as it is reflected in the lengths of those arguments); but permitted
4. to produce results that share structure with their arguments.

It is usually taken for granted that standard Scheme lists have these properties. This SRFI explicitly specifies that random-access lists share them.

syntax: `(quote datum)`

Syntax: <Datum> should be a syntactic datum.

Semantics: `(quote <datum>)` evaluates to the datum value represented by <datum> (see section 4.3 of R6RS). This notation is used to include constants.

When the datum value represented by <datum> contains pair structure, `quote` produces random-access pairs.

```
(quote a)                                    a
(quote #(a b c))                             #(a b c)
(quote (+ 1 2))                              (+ 1 2)
```

As noted in section 4.3.5 of R6RS, `(quote <datum>)` may be abbreviated as `'<datum>`:

```
'"abc"                                       "abc"
'145932                                      145932
'a                                           a
'#(a b c)                                    #(a b c)
'()                                          ()
'(+ 1 2)                                     (+ 1 2)
'(quote a)                                   (quote a)
''a                                          (quote a)
```

As noted in section 5.10 of R6RS, constants are immutable.

> *Note:* Different constants that are the value of `quote` expression may share the same locations.

procedure: `(equal? obj1 obj2) → bool`

The `equal?` predicate returns `#t` if and only if the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees.

The `equal?` predicate treats pairs and vectors as nodes with outgoing edges, uses `string=?` to compare strings, uses `bytevector=?` to compare bytevectors, and uses `eqv?` to compare other nodes.

```
(equal? 'a 'a)                                  #t
(equal? '(a) '(a))                              #t
(equal? '(a (b) c)
        '(a (b) c))                             #t
(equal? "abc" "abc")                            #t
(equal? 2 2)                                    #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))                     #t
(equal? '#vu8(1 2 3 4 5)
        (u8-list->bytevector
         '(1 2 3 4 5))                          #t
(equal? (lambda (x) x)
        (lambda (y) y))                         unspecified

(let* ((x (list 'a))
       (y (list 'a))
       (z (list x y)))
  (list (equal? z (list y x))
        (equal? z (list x x))))                 (#t #t)
```

procedure: `(pair? obj) → bool`

Returns `#t` if *obj* is a pair, and otherwise returns `#f`. This operation must take *O(1)* time.

```
(pair? '(a . b))                                #t
(pair? '(a b c))                                #t
(pair? '())                                     #f
(pair? '#(a b))                                 #f
```

procedure: `(cons obj1 obj2) → pair`

Returns a newly allocated pair whose car is obj_1 and whose cdr is obj_2. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object. This operation must take *O(1)* time.

```
(cons 'a '())                                    (a)
(cons '(a) '(b c d))                             ((a) b c d)
(cons "a" '(b c))                                ("a" b c)
(cons 'a 3)                                      (a . 3)
(cons '(a b) 'c)                                 ((a b) . c)
```

procedure: `(car pair) → obj`

Returns the contents of the car field of *pair*. This operation must take *O(1)* time.

```
(car '(a b c))                                   a
(car '((a) b c d))                               (a)
(car '(1 . 2))                                   1
(car '())                               &assertion exception
```

procedure: `(cdr pair) → obj`

Returns the contents of the cdr field of *pair*. This operation must take *O(1)* time.

```
(cdr '((a) b c d))                          (b c d)
(cdr '(1 . 2))                              2
(cdr '())                           &assertion exception
```

procedure: `(caar pair) → obj`
procedure: `(cadr pair) → obj`
…
procedure: `(cdddar pair) → obj`
procedure: `(cddddr pair) → obj`

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by
`(define caddr (lambda (x) (car (cdr (cdr x))))).`

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all. These operations must take *O(1)* time.

procedure: `(null? obj) → bool`

Returns `#t` if *obj* is the empty list, `#f` otherwise. This procedure is equivalent to the `null?` procedure of the R6RS base library.

procedure: `(list? obj) → bool`

Returns `#t` if *obj* is a list, `#f` otherwise. By definition, all lists are chains of pairs that have finite length and are terminated by the empty list. This operation must take time bounded by *O(log(n))*, where *n* is the number of pairs in the chain forming the potential list.

```
(list? '(a b c))                            #t
(list? '())                                 #t
(list? '(a . b))                            #f
```

procedure: `(list obj ...) → list`

Returns a newly allocated list of its arguments. This operation must take time bounded by *O(n)*, where *n* is the number of arguments to `list`.

```
(list 'a (+ 3 4) 'c)                        (a 7 c)
(list)                                      ()
```

procedure: `(make-list k) → list`
procedure: `(make-list k obj) → list`

Returns a newly allocated list of *k* elements. If a second argument is given, then each element is initialized to *obj*. Otherwise the initial contents of each element is unspecified. This operation must take time and space bounded by *O(log(k))*.

```
(make-list 5 0)                             (0 0 0 0 0)
```

procedure: `(length list) → k`

Returns the length of *list*. This operation must take time bounded by *O(log(n))*, where *n* is the length of the list.

```
(length '(a b c))                           3
(length '(a (b) (c)))                       3
(length '())                                0
```

procedure: `(length<=? obj k) → bool`

Returns true if *obj* is a chain of at least *k* pairs and false otherwise. This operation must take time bounded by *O(log(min(k,n)))*, where *n* is the length of the chain of pairs.

```
(length<=? 'not-a-list 0)                         #t
(length<=? '(a . b) 0)                            #t
(length<=? '(a . b) 1)                            #t
(length<=? '(a . b) 2)                            #f
```

procedure: `(append list ... obj) → obj`

Returns a chain of pairs consisting of the elements of the first *list* followed by the elements of the other lists, with *obj* as the cdr of the final pair. An improper list results if *obj* is not a list. This operation must take time bounded by *O(log(n))*, where *n* is the total number of elements in the given lists.

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

procedure: `(reverse list) → list`

Returns a newly allocated list consisting of the element of *list* in reverse order. This operation must take time bounded by *O(n)* where *n* is the length of the list.

```
(reverse '(a b c))                                (c b a)
(reverse '(a (b c) 'd '(e (f))))                  ((e (f)) d (b c) a)
```

procedure: `(list-tail obj k) → obj`

Obj should be a chain of pairs with a count of at least *k*. The `list-tail` procedure returns the object obtained by omitting the first *k* elements in *obj*. This operation must take time bounded by *O(log(min(k,n)))*, where *n* is the length of the chain of pairs.

```
(list-tail '(a b c d) 0)                          (a b c d)
(list-tail '(a b c d) 2)                          (c d)
(list-tail 'not-a-list 0)                         not-a-list
```

Implementation responsibilities: The implementation must check that *obj* is a chain of pairs whose count is at least *k*.

procedure: `(list-ref pair k) → obj`

Pair must be a chain of pairs whose count is at least *k + 1*. The `list-ref` procedure returns the *k*th element of *pair*. This operation must take time bounded by *O(min(k,log(n)))*, where *n* is the length of the chain of pairs.

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

Implementation responsibilities: The implementation must check that *pair* is a chain of pairs whose count is at least *k + 1*.

procedure: `(list-set pair k obj) → obj`

Pair must be a chain of pairs whose count is at least *k + 1*. The `list-set` procedure returns the chain of pairs obtained by replacing the *k*th element with *obj*. This operation must take time bounded by *O(min(k,log(n)))*, where *n* is the length of the chain of pairs.

```
(list-set '(a b c d) 2 'x)                        (a b x d)
```

Implementation responsibilities: The implementation must check that *pair* is a chain of pairs whose count is at least *k + 1*.

procedure: `(list-ref/update pair k proc) → obj1 obj2`

Returns the same results as:

```
(values (list-ref pair k)
        (list-set pair k (proc (list-ref pair k))))
```

but it may be implemented more efficiently.

```
(list-ref/update '(7 8 9 10) 2 -)             9 (7 8 -9 10)
```

procedure: `(map proc list1 list2 ...) → list`

The *list*s should all have the same length. *Proc* should accept as many arguments as there are *list*s and return a single value.

The `map` procedure applies *proc* element-wise to the elements of the *list*s and returns a list of the results, in order. *Proc* is always called in the same dynamic environment as `map` itself. The order in which *proc* is applied to the elements of the *list*s 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)
```

Implementation responsibilities: The implementation should check that the *list*s all have the same length. The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

procedure: `(for-each proc list1 list2 ...) → unspecified`

The lists should all have the same length. *Proc* should accept as many arguments as there are lists.

The `for-each` procedure applies *proc* element-wise to the elements of the *list*s for its side effects, in order from the first element to the last. *Proc* is always called in the same dynamic environment as `for-each` itself. The return values of `for-each` are 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)

(for-each (lambda (x) x) '(1 2 3 4))          unspecified
(for-each even? '())                          unspecified
```

Implementation responsibilities: The implementation should check that the *list*s all have the same length. The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

> *Note:* Implementations of `for-each` may or may not tail-call *proc* on the last element.

Representation conversion

procedure: `(random-access-list->linear-access-list ra-list) → la-list`
procedure: `(linear-access-list->random-access-list la-list) → ra-list`

These procedures convert between (potentially) distinct representations of lists. To avoid confusion, parameters named *ra-list* range over lists represented with random-access lists, i.e. objects satisfying the `list?` predicate described above, while parameters named *la-list* range over lists represented with the more traditional linear-access lists, i.e. objects satisfying the `list?` predicate of R6RS. In systems that represent all lists as random-access lists, these conversions may simply be list identity procedures.

Implementation requirements

Random-access pairs must be disjoint from all other base types with the possible exception of (linear-access) pairs.

The external representation of random-access pairs is unspecified. The behavior of `equal?` when given a random-access pair and a sequential-access pair is unspecified in implementations with disjoint representations.

The behavior of `eq?` and `eqv?` on random-access pairs must be the same as that for pairs, vectors, or records. Namely, two random-access pair objects are `eq?` if and only if they are `eqv?`, and they are `eqv?` if and only if they refer to the same location in the store.

All argument checking for each operation must be done within the time bounds given for that operation.

Implementations are encouraged, but not required, to support random-access pairs and lists as their primary pair and list representation. In such an implementation, the external representation of random-access pairs and list should be as described in section 4.3.2 (Pairs and lists) of R^{6}RS, the behavior of equivalence predicates on random-access pairs should be as described in section 11.5 (Equivalence predicates) of R^{6}RS, and so on. In short, all pairs should be random-access pairs.

Implementations supporting SRFI Libraries [4] and SRFI 101 must provide the following libraries:

```
(srfi :101)                               ; Composite libraries
(srfi :101 random-access-lists)

(srfi :101 random-access-lists procedures)  ; Procedures only
(srfi :101 random-access-lists syntax)      ; Syntax only
```

References

[1] **Purely Functional Random-Access Lists** Chris Okasaki, *Functional Programming Languages and Computer Architecture*, June 1995, pages 86-95.
`http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#fpca95`
[2] **Revised^6 Report on the Algorithmic Language Scheme** Michael Sperber, *et al.* (Editors)
`http://www.r6rs.org/`
[3] **Revised^6 Report on the Algorithmic Language Scheme, Standard Libraries** Michael Sperber, *et al.* (Editors)
`http://www.r6rs.org/`
[4] **SRFI 97: SRFI Libraries** David Van Horn
`http://srfi.schemers.org/srfi-97/`
[5] **PLaneT: Purely Functional Random-Access Lists** David Van Horn
`http://planet.plt-scheme.org/display.ss?package=ralist.plt&owner=dvanhorn`

Editor: Donovan Kolbly

Deques

[**Red Edition item 10**] What immutable deque library should R7RS-large provide?

SRFI 134 is a library for immutable double-ended queues, with O(1) addition and removal time at both ends.

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 TODO names of the procedures specified here ends with an exclamation point.

The names described in this section comprise the `(scheme ideque)` library.

Procedure index

Constructors: `ideque, ideque-tabulate, ideque-unfold, ideque-unfold-right`

Predicates: `ideque?, ideque-empty?, ideque=, ideque-any, ideque-every`

Queue operations: `ideque-front, ideque-back, ideque-remove-front, ideque-remove-back, ideque-add-front, ideque-add-back`

Other accessors: `ideque-ref, ideque-take, ideque-take-right, ideque-drop, ideque-drop-right, ideque-split-at`

The whole ideque: `ideque-length, ideque-append, ideque-reverse, ideque-count, ideque-zip`

Mapping: `ideque-map, ideque-filter-map, ideque-for-each, ideque-for-each-right, ideque-fold, ideque-fold-right, ideque-append-map`

Filtering: `ideque-filter, ideque-remove, ideque-partition`

Searching: `ideque-find, ideque-find-right, ideque-take-while, ideque-take-while-right, ideque-drop-while, ideque-drop-while-right, ideque-span, ideque-break`

Conversion: `list->ideque, ideque->list, generator->ideque, ideque->generator`

We specify required time efficiency upper bounds using big-O notation. We note when, in some cases, there is “slack” between the required bound and the theoretically optimal bound for an operation. Implementations may use data structures with amortized time bounds, but should document which bounds hold in only an amortized sense.

Deques are disjoint from all other Scheme types.

Constructors

`(ideque` *element …*`)` → *ideque???* ideque library procedure

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*`)` → *ideque???* ideque library procedure

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*`)` → *ideque???* ideque library procedure

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*`)` → *ideque???* ideque library procedure

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*`)` → *boolean???* ideque library procedure

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

`(ideque-empty?` *ideque*`)` → *boolean???* ideque library procedure

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

`(ideque=` *elt= ideque …*`)` → *boolean???* ideque library procedure

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 $ideque_i$, then its second argument is from $ideque_{i+1}$, 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 $ideque_i$ is compared to $ideque_{i+1}$ (as opposed, for example, to comparing $ideque_1$ to every $ideque_i$, 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*`)` → *value or boolean???* ideque library procedure
`(ideque-every` *pred ideque*`)` → *ideque???* ideque library procedure

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*`)` → *value???* ideque library procedure
`(ideque-back` *ideque*`)` → *value???* ideque library procedure

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???* ideque library procedure
`(ideque-remove-back` *ideque*`)` → *ideque???* ideque library procedure

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???* ideque library procedure
`(ideque-add-back` *ideque obj*`)` → *ideque???* ideque library procedure

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

Other accessors

`(ideque-ref` *ideque n*`)` → *value???* ideque library procedure

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???* ideque library procedure
`(ideque-take-right` *ideque n*`)` → *ideque???* ideque library procedure

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???* ideque library procedure
`(ideque-drop-right` *ideque n*`)` → *ideque???* ideque library procedure

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*`)` → *ideque??? ideque???* ideque library procedure

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*`)` → *integer???* ideque library procedure

Returns the length of *ideque* as an exact integer. May take $O(n)$ time, though $O(1)$ is optimal. TODO "may take $O(1)$ in the best case"??? I thought we were giving average cases.

`(ideque-append` *ideque* …`)` → *ideque???* ideque library procedure

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.~~ some implementations may only require $O(k \log n)$. TODO I assume that's "lg", not that it really matters.

`(ideque-reverse` *ideque*`)` → *ideque???* ideque library procedure

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

`(ideque-count` *pred ideque*`)` → *integer???* ideque library procedure

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` *ideque*$_1$ *ideque*$_2$ …`)` → *ideque???* ideque library procedure

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.

Mapping

`(ideque-map` *proc ideque*`)` → *ideque???* ideque library procedure

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*`)` → *ideque???* ideque library procedure

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*`)` → *unspecified???* ideque library procedure
`(ideque-for-each-right` *proc ideque*`)` → *unspecified???* ideque library procedure

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*`)` → *value???* ideque library procedure
`(ideque-fold-right` *proc nil ideque*`)` → *value???* ideque library procedure

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*`)` → *ideque???* ideque library procedure

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.

Filtering

`(ideque-filter` *pred ideque*`)` → *ideque???* ideque library procedure
`(ideque-remove` *pred ideque*`)` → *ideque???* ideque library procedure

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

TODO In the same order?

`(ideque-partition` *proc ideque*`)` → *ideque??? ideque???* ideque library procedure

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.

Searching

`(ideque-find` *pred ideque failure*`)` → *value???* ideque library procedure
`(ideque-find` *pred ideque*`)` → *value???* ideque library procedure
`(ideque-find-right` *pred ideque failure*`)` → *value???* ideque library procedure
`(ideque-find-right` *pred ideque*`)` → *value???* ideque library procedure

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???* ideque library procedure
`(ideque-take-while-right` *pred ideque*`)` → *ideque???* ideque library procedure

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???* ideque library procedure
`(ideque-drop-while-right` *pred ideque*`)` → *ideque???* ideque library procedure

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??? ideque???* ideque library procedure
`(ideque-break` *pred ideque*`)` → *ideque??? ideque???* ideque library procedure

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.

TODO Explain why the 2 names.

Conversion

`(list->ideque` *list*`)` → *ideque???* ideque library procedure
`(ideque->list` *ideque*`)` → *list???* ideque library procedure

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???* ideque library procedure
`(ideque->generator` *ideque*`)` → *generator???* ideque library procedure

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.
Editor: Arthur A. Gleckler

Immutable texts

[**Red Edition item 11**] What immutable text library should R7RS-large provide?

> SRFI 135 is a library for immutable character sequences with O(1) access time. Its procedures accept strings as well as texts, so that a separate string library may not be as useful.

TODO This material and the “Basic Concepts” stuff further down should be reorganized. Some material will go to the Rationale, but much of it should stay.

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.

Issues

None.

Procedure Index

Here is a list of the procedures provided by this SRFI:

Predicates `text?` `textual?`

```
textual-null?
textual-every        textual-any
```

Constructors `make-text` `text`

```
text-tabulate
text-unfold          text-unfold-right
```

Conversion `textual->text`

```
textual->string      textual->vector        textual->list
string->text         vector->text           list->text
reverse-list->text
textual->utf8        textual->utf16be
textual->utf16       textual->utf16le
utf8->text           utf16be->text
utf16->text          utf16le->text
```

Selection `text-length` `textual-length`

```
text-ref             textual-ref
subtext              subtextual
textual-copy
textual-take         textual-take-right
textual-drop         textual-drop-right
textual-pad          textual-pad-right
textual-trim         textual-trim-right     textual-trim-both
```

Replacement `textual-replace`

Comparison `textual=?` `textual-ci=?`

```
textual<?            textual-ci<?
textual>?            textual-ci>?
textual<=?           textual-ci<=?
textual>=?           textual-ci>=?
```

Prefixes & suffixes `textual-prefix-length` `textual-suffix-length`

```
textual-prefix?      textual-suffix?
```

Searching `textual-index` `textual-index-right`

```
textual-skip         textual-skip-right
textual-contains     textual-contains-right
```

Case conversion `textual-upcase` `textual-downcase`

```
textual-foldcase     textual-titlecase
```

Concatenation `textual-append` `textual-concatenate` `textual-concatenate-reverse`

```
textual-join
```

Fold & map & friends `textual-fold` `textual-fold-right`

```
textual-map          textual-for-each
textual-map-index    textual-for-each-index
textual-count
textual-filter       textual-remove
```

Replication & splitting `textual-replicate` `textual-split`

Immutable texts

Basic concepts

The names described in this section comprise the `(scheme text)` library.

Conceptual model

Immutable texts are like strings except they can't be mutated.

Immutability makes it easier to use space-efficient representations such as UTF-8 and UTF-16 without incurring the cost of scanning from the beginning when character indexes are used (as with `string-ref`).

When mutation is not needed, immutable texts are likely to be more efficient than strings with respect to space or time. In some implementations, immutable texts may be more efficient than strings with respect to both space and time.

Subtypes

This SRFI defines two new types:

- *text* is a type consisting of the immutable texts for which `text?` returns true.
- *textual* is a union type consisting of the texts and strings for which `textual?` returns true.

The subtypes of the new *textual* type include the new *text* type and Scheme's traditional *string* type, which consists of the values for which `string?` returns true. The *string* type includes both mutable strings and the (conceptually) immutable strings that are the values of string literals and calls to `symbol->string`.

Implementations of this SRFI are free to extend the *textual* type by adding new subtypes, provided all procedures whose names begin with `textual-` are extended to accept values of those new subtypes. Implementations of this SRFI should not extend the *text* type unless its extended values are immutable, are accepted as texts by all procedures of this SRFI (including the `text?` predicate), and achieve the performance required by this SRFI with respect to both time and space.

External representation

This SRFI does not require any particular external representation for immutable texts, but recommends immutable texts have almost the same external representation as strings, substituting Unicode's left-pointing and right-pointing double angle quotation marks (« and », code points `#xab` and `#xbb`) for the double quotes that delimit strings, and allowing those double angle quotation marks to be escaped within the external representations of both texts and strings. That external representation is used by this SRFI's examples.

TODO This requires adding the guillemets « and » to Scheme's character set. How is this going to be done?

When feasible, implementations of this SRFI should also consider:

- extending the `equal?` procedure to regard two immutable texts t1 and t2 as equal if and only if `(textual=? t1 t2)`, while regarding an immutable text as unequal to anything that isn't an immutable text.
- extending the `display` procedure to accept immutable texts, treating them the same as strings;
- extending the `write` procedure to generate the external syntax recommended for immutable texts;
- extending the `read` procedure to accept the external syntax recommended for immutable texts;
- extending interpreters and compilers to accept quoted literals expressed using the external syntax recommended for immutable texts; R7RS section 4.1.2 mandates this extension if `read` is extended to accept the external syntax for texts.

Note: Those extensions cannot be implemented portably, so portable code should not rely on them. TODO As I see it, this SRFI as part of the large language means that the above features must be redefined in R7RS-Small. Same with the guillemets above. Either that, or the recommendations given here should be removed.

Textual input and output ports

Textual input and output ports analogous to string input and output ports would be nice, but they too cannot be implemented portably. Leaving them for another SRFI allows all of this SRFI to be implemented portably with reasonable efficiency.

TODO Replace with "implementations are encouraged, but not required, to provide textual I/O ports analogous to string I/O ports"?

Shared storage

All strings and other mutable objects returned by the procedures specified within this SRFI are newly allocated and may be mutated with abandon.

No externally visible string ever shares storage with any text. All strings and other mutable objects passed to the procedures specified within this SRFI may be mutated without affecting the behavior of any text.

The immutability of texts allows sharing of substructure, so `subtext`, `textual-append`, and similar operations can be faster and more space-efficient than Scheme's `substring` and `string-append` operations.

Although distinct texts may share storage internally, this is undetectable because texts are immutable and the procedures that operate on texts do not directly expose any of their internal components.

Implementations that share storage between texts must satisfy the following requirement: There is some reasonably 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.

Example: For the sample implementations with their default configurations, the worst case arises with UTF-8, when a 1-character ASCII text retains up to 127 characters of a text that is no longer reachable, and all 127 of those retained characters lie outside Unicode's Basic Multilingual Plane (BMP). Making reasonable assumptions about the representations of records, vectors, bytevectors, and strings on a 64-bit machine, that shared text would occupy no more than about 16 times the space occupied by an unshared representation. If the retained characters were in the BMP, the shared text would occupy no more than about 8 times the space occupied by an unshared representation. If the retained characters were ASCII, the shared text would occupy no more than about 4 times the space occupied by an unshared representation. The sample implementations can be configured to reduce those worst-case bounds, most obviously by reducing the maximum number of characters that can be shared with a very short text.

Naming conventions

The procedures of this SRFI follow a consistent naming scheme, and are consistent with the conventions developed in SRFI 1 and used in SRFI 13 and SRFI 130. Indeed, most of the names specified here were derived from SRFI 130's names by replacing `string` with `text` or `textual`. As in SRFI 130, procedures that have left/right directional variants use no suffix to specify left-to-right operation, `-right` to specify right-to-left operation, and `-both` to specify both.

Note, however, that `textual-index`, `textual-index-right`, `textual-skip`, and `textual-skip-right`, return `#f` when no match is found. In SRFI 130, their analogues always return cursors.

The order of common arguments is consistent across the different procedures.

For convenience, most procedures that accept a text as argument will also accept a string. When given a string, those procedures behave as though the string is first converted to a text, so passing a text is likely to be more efficient than passing a string.

Performance requirements

A few procedures are required to execute in O(1) time: `text?`, `textual?`, `text-length`, and `text-ref`.

If the first two arguments passed to `textual-contains` and `textual-contains-right` are texts, then those procedures must run in O(m n) time, where m and n are the lengths of the two subtexts specified by their arguments. If either of the first two arguments is a string, there is no such requirement.

The other procedures specified by this SRFI should run in amortized linear time, not counting time spent in procedures and predicates that were passed as arguments. That is not an absolute requirement, but the sample implementations are designed to deliver that level of performance for most procedures provided none of their textual arguments are strings. When strings are passed as arguments, the running time is unlikely to be linear unless `string-ref` runs in constant time, and that is not required by any of the Scheme standards.

Indeed, this SRFI was designed to make efficient text processing easier in systems whose `string-ref` procedure does not run in constant time. For efficiency, portable code should use strings only for fairly short sequences of characters. Representations with guaranteed efficiency (such as the immutable texts of this SRFI) should be used for longer texts.

Note: A procedure that runs in O(1) time does not necessarily take the same time for all inputs. Furthermore O(1) = O(1000), so procedures that run in O(1) time can still be quite slow. The `text-ref` procedure, for example, may have worst cases for which it is hundreds of times slower than `text?`. Even the average case for `text-ref` is likely to be several times as slow as the worst case for `text?`.

Unicode

TODO This section is about UTF specifically, rather than Unicode. I think this could be substantially condensed, with a paragraph on the various UTF formats, followed by the recommendation to use the portable solution below.

During the early development of Unicode, its designers believed a 16-bit character set would suffice, which is why Java's `char` type has only 16 bits. When Unicode expanded to 1114112 code points, 16 bits were no longer enough to encode all Unicode characters.

The Unicode standard defines three encoding forms for arbitrary sequences of Unicode characters:

UTF-32 is a fixed-width encoding in which every character is represented by a straightforward 32-bit representation of its code point.

UTF-16 is a variable-width encoding in which the most common characters are represented by 16-bit representations of their code points, but characters outside the Basic Multilingual Plane (BMP) are represented by a surrogate pair consisting of two consecutive 16-bit code units.

UTF-8 is a variable-width encoding in which ASCII characters are represented by 8-bit representations of their code points, but other characters are encoded by a sequence of two, three, or four 8-bit code units.

UTF-32 is a convenient internal representation and is used as such by several string libraries for C, C++, and Python, but it is the least compact of the three representations and is seldom used in files. UTF-16 is convenient for applications that use only the BMP, and supports fast sequential processing of arbitrary Unicode; variants of UTF-16 are used by Windows for files and by Java and C# as an internal representation. UTF-8 is upwardly compatible with the ASCII encoding and supports fast sequential processing of arbitrary Unicode; it is widely used for files on non-Windows machines and is also used by some C libraries.

The Scheme programming language does not expose the internal representation of strings. Some implementations of Scheme use UTF-32 or a similar encoding, which makes `string-length`, `string-ref`, and `string-set!` run in O(1) time. Some implementations of Scheme use UTF-16, which saves space at the expense of making `string-ref` take time proportional to the length of a string. Some implementations of Scheme use UTF-8, which saves even more space for ASCII strings while making `string-ref` run in linear time.

Although Scheme's string data type allows portable code to use strings independently of their internal representation, the variation in performance between implementations has created a problem for programs that use long strings. In some systems, long strings are inefficient with respect to space; in other systems, long strings are inefficient with respect to time.

The portable solution to this dilemma is to use Scheme's mutable strings only for buffers and other relatively short sequences of characters, while using the immutable texts defined by this SRFI for long sequences of characters.

Note: SRFI 130 suggests an alternative solution: Portable code should process strings sequentially using cursors instead of indexes, and should avoid mutation of strings by using vectors of characters instead, while hoping all major implementations of Scheme will soon convert their strings to use compact internal representations such as UTF-8 or UTF-16. That hope is unlikely to be realized, because a lot of legacy code assumes `string-ref` runs in O(1) time, as recommended by the R6RS, and mutable strings represented in UTF-32 or similar are more efficient than vectors of characters with respect to both time and space. At present, several implementations of Scheme support Unicode while providing `string-ref` and `string-set!` procedures that run in O(1) time; making those operations run asymptotically slower would displease some users of those systems.

Notation

In the following procedure specifications:

- A text argument is an immutable text.
- A textual argument is an immutable text or a string.
- A char argument is a character.
- An idx argument is an exact non-negative integer specifying a valid character index into a text or string. The valid character indexes of a text or string textual of length n are the exact integers idx satisfying 0 <= idx < n.
- A k argument or result is a *position*: an exact non-negative integer that is either a valid character index for one of the textual arguments or is the length of a textual argument.
- start and end arguments are positions specifying a half-open interval of indexes for a subtext or substring. When omitted, start defaults to 0 and end to the length of the corresponding textual argument. It is an error unless 0 <= start <= end <= `(textual-length textual)`; the sample implementations detect that error and raise an exception.
- A len or nchars argument is an exact non-negative integer specifying some number of characters, usually the length of a text or string.
- A pred argument is a unary character predicate, taking a character as its one argument and returning a value that will be interpreted as true or false. Unless noted otherwise, as with `textual-every` and `textual-any`, all predicates passed to procedures specified in this SRFI may be called in any order and any number of times. It is an error if pred has side effects or does not behave functionally (returning the same result whenever it is called with the same character); the sample implementations do not detect those errors.
- An obj argument may be any value at all.

It is an error to pass values that violate the specification above.

Arguments given in square brackets are optional. Unless otherwise noted in the text describing the procedure, any prefix of these optional arguments may be supplied, from zero arguments to the full list. When a procedure returns multiple values, this is shown by listing the return values in square brackets, as well. So, for example, the procedure with signature

```
halts? f [x init-store] → [boolean integer]
```

would take one (f), two (f, x) or three (f, x, init-store) input arguments, and return two values, a boolean and an integer.

An argument followed by "..." means zero or more elements. So the procedure with the signature

```
sum-squares x ...  → number
```

takes zero or more arguments (x ...), while the procedure with signature

```
spell-check doc dict1 dict2 ... → string-list
```

takes two required arguments (doc and dict$_1$) and zero or more optional arguments (dict$_2$ ...).

If a procedure's return value is said to be "unspecified," the procedure returns a single result whose value is unconstrained and might even vary from call to call.

Predicates

`(text?` *obj*`)` → *boolean* text library procedure

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*`)` → *boolean* text library procedure

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

`(textual-null? ` *text*`)` → *boolean* — text library procedure

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

`(textual-every ` *pred textual start end*`)` → *value* — text library procedure
`(textual-every ` *pred textual*`)` → *value* — text library procedure
`(textual-any ` *pred textual start end*`)` → *value* — text library procedure
`(textual-any ` *pred textual*`)` → *value* — text library procedure

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`).

Constructors

`(make-text ` *len char*`)` → *text* — text library procedure

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

`(text ` *char …*`)` → *text* — text library procedure

Returns a text consisting of the given characters.

`(text-tabulate ` *proc len*`)` → *text* — text library procedure

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*`)` → *text* — text library procedure
`(text-unfold ` *stop? mapper successor seed*`)` → *text* — text library procedure

This is a fundamental constructor for texts.

- successor is used to generate a series of "seed" values from the initial seed:
  seed, (successor seed), (successor^2 seed), (successor^3 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*`)` → *text* text library procedure
`(text-unfold-right` *stop? mapper successor seed*`)` → *text* text library procedure

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 »
```

Conversion

`(textual->text` *textual*`)` → *text* text library procedure

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*`)` → *string* text library procedure
`(textual->string` *textual*`)` → *string* text library procedure
`(textual->vector` *textual start end*`)` → *char-vector* text library procedure
`(textual->vector` *textual*`)` → *char-vector* text library procedure
`(textual->list` *textual start end*`)` → *char-list* text library procedure
`(textual->list` *textual*`)` → *char-list* text library procedure

`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*`)` → *text* text library procedure
`(string->text` *string*`)` → *text* text library procedure
`(vector->text` *char-vector start end*`)` → *text* text library procedure
`(vector->text` *char-vector*`)` → *text* text library procedure
`(list->text` *char-list start end*`)` → *text* text library procedure
`(list->text` *char-list*`)` → *text* text library procedure

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*`)` → *text* text library procedure

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

```
(reverse-list->text '(#\a #\B #\c))}{procedure}{«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*`)` → *bytevector* text library procedure
`(textual->utf8` *textual*`)` → *bytevector* text library procedure
`(textual->utf16` *textual start end*`)` → *bytevector* text library procedure
`(textual->utf16` *textual*`)` → *bytevector* text library procedure
`(textual->utf16be` *textual start end*`)` → *bytevector* text library procedure
`(textual->utf16be` *textual*`)` → *bytevector* text library procedure
`(textual->utf16le` *textual start end*`)` → *bytevector* text library procedure
`(textual->utf16le` *textual*`)` → *bytevector* text library procedure

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*`)` → *text* text library procedure
`(utf8->text` *bytevector*`)` → *text* text library procedure
`(utf16->text` *bytevector start end*`)` → *text* text library procedure
`(utf16->text` *bytevector*`)` → *text* text library procedure
`(utf16be->text` *bytevector start end*`)` → *text* text library procedure
`(utf16be->text` *bytevector*`)` → *text* text library procedure
`(utf16le->text` *bytevector start end*`)` → *text* text library procedure
`(utf16le->text` *bytevector*`)` → *text* text library procedure

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.

Selection

`(text-length` *text*`)` → *len* text library procedure

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

`(text-ref` *text idx*`)` → *char* text library procedure

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

`(textual-length` *textual*`)` → *len* text library procedure
`(textual-ref` *textual idx*`)` → *char* text library procedure

`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*`)` → *text* text library procedure
`(subtextual` *textual start end*`)` → *text* text library procedure

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@)*`)` → *text* text library procedure

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*`)` → *text* text library procedure
`(textual-drop` *textual nchars*`)` → *text* text library procedure
`(textual-take-right` *textual nchars*`)` → *text* text library procedure
`(textual-drop-right` *textual nchars*`)` → *text* text library procedure

`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*`)` → *text* text library procedure
`(textual-pad` *textual len*`)` → *text* text library procedure
`(textual-pad-right` *textual len char start end*`)` → *text* text library procedure
`(textual-pad-right` *textual len*`)` → *text* text library procedure

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 `#\space`.

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*`)` → *text* text library procedure
`(textual-trim` *textual*`)` → *text* text library procedure
`(textual-trim-right` *textual pred start end*`)` → *text* text library procedure
`(textual-trim-right` *textual*`)` → *text* text library procedure
`(textual-trim-both` *textual pred start end*`)` → *text* text library procedure
`(textual-trim-both` *textual*`)` → *text* text library procedure

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,»
```

Replacement

`(textual-replace` *textual1 textual2 start1 end1 start2 end2*`)` → *text* text library procedure
`(textual-replace` *textual1 textual2 start1 end1*`)` → *text* text library procedureReturns

```
(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.»
```

Comparison

`(textual=?` *textual1 textual2 textual3 …* `)` → *boolean* text library procedure

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 …*`)` → *boolean* text library procedure
`(textual>?` *textual1 textual2 textual3 …*`)` → *boolean* text library procedure
`(textual<=?` *textual1 textual2 textual3 …*`)` → *boolean* text library procedure
`(textual>=?` *textual1 textual2 textual3 …*`)` → *boolean* text library procedure

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 …*) → *boolean* text library procedure

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 …*) → *boolean* text library procedure
(`textual-ci>?` *textual1 textual2 textual3 …*) → *boolean* text library procedure
(`textual-ci<=?` *textual1 textual2 textual3 …*) → *boolean* text library procedure
(`textual-ci>=?` *textual1 textual2 textual3 …*) → *boolean* text library procedure

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

Prefixes & suffixes

(`textual-prefix-length` *textual1 textual2 start1 end1 start2 end2*) → *integer* text library procedure
(`textual-prefix-length` *textual1 textual2*) → *integer* text library procedure
(`textual-suffix-length` *textual1 textual2 start1 end1 start2 end2*) → *integer* text library procedure
(`textual-suffix-length` *textual1 textual2*) → *integer* text library procedure

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*) → *boolean* text library procedure
(`textual-prefix?` *textual1 textual2*) → *boolean* text library procedure
(`textual-suffix?` *textual1 textual2 start1 end1 start2 end2*) → *boolean* text library procedure
(`textual-suffix?` *textual1 textual2*) → *boolean* text library procedure

Is textual1 a prefix/suffix of textual2?

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

Searching

(`textual-index` *textual pred start end*) → *idx-or-false* text library procedure
(`textual-index` *textual pred*) → *idx-or-false* text library procedure
(`textual-index-right` *textual pred start end*) → *idx-or-false* text library procedure
(`textual-index-right` *textual pred*) → *idx-or-false* text library procedure
(`textual-skip` *textual pred start end*) → *idx-or-false* text library procedure
(`textual-skip` *textual pred*) → *idx-or-false* text library procedure
(`textual-skip-right` *textual pred start end*) → *idx-or-false* text library procedure
(`textual-skip-right` *textual pred*) → *idx-or-false* text library procedure

`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*`)` → *idx-or-false* text library procedure
`(textual-contains` *textual1 textual2*`)` → *idx-or-false* text library procedure
`(textual-contains-right` *textual1 textual2 start1 end1 start2 end2*`)` → *idx-or-false* text library procedure
`(textual-contains-right` *textual1 textual2*`)` → *idx-or-false* text library procedure

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.

Case conversion

`(textual-upcase` *textual*`)` → *text* text library procedure
`(textual-downcase` *textual*`)` → *text* text library procedure
`(textual-foldcase` *textual*`)` → *text* text library procedure
`(textual-titlecase` *textual*`)` → *text* text library procedure

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.

Concatenation

`(textual-append` *textual …*`)` → *text* text library procedure

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

`(textual-concatenate` *textual-list*`)` → *text* text library procedure

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*`)` → *text* text library procedure
`(textual-concatenate-reverse` *textual-list*`)` → *text* text library procedure

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*`)` → *text* text library procedure
`(textual-join` *textual-list*`)` → *text* text library procedure

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)) => «:»
```

Fold & map & friends

(`textual-fold` *kons knil textual start end*) → *value* text library procedure
(`textual-fold` *kons knil textual*) → *value* text library procedure
(`textual-fold-right` *kons knil textual start end*) → *value* text library procedure
(`textual-fold-right` *kons knil textual*) → *value* text library procedure

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 ...*) → *text* text library procedure

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 …*`)` → *unspecified* text library procedure

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*`)` → *text* text library procedure
`(textual-map-index` *proc textual*`)` → *text* text library procedure

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*`)` → *unspecified* text library procedure
`(textual-for-each-index` *proc textual*`)` → *unspecified* text library procedure

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*`)` → *integer* text library procedure
`(textual-count` *textual pred*`)` → *integer* text library procedure

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*`)` → *text* text library procedure
`(textual-filter` *pred textual*`)` → *text* text library procedure
`(textual-remove` *pred textual start end*`)` → *text* text library procedure
`(textual-remove` *pred textual*`)` → *text* text library procedure

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.

Replication & splitting

`(textual-replicate` *textual from to start end*`)` → *text* text library procedure
`(textual-replicate` *textual from to*`)` → *text* text library procedure

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*`)` → *list* text library procedure
`(textual-split` *textual delimiter*`)` → *list* text library procedure

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 `"\r\n"`. 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.

References & links

[**CommonLisp**] *Common Lisp: the Language.*
Guy L. Steele Jr. (editor).
Digital Press, Maynard, Mass., second edition 1990.
Available at `http://www.elwood.com/alu/table/references.htm#cltl2`.

The Common Lisp "HyperSpec," produced by Kent Pitman, is essentially the ANSI spec for Common Lisp: `http://www.harlequin.com/education/books/HyperSpec/`.

[**MIT-Scheme**] `http://www.swiss.ai.mit.edu/projects/scheme/`

[**R5RS**] Revised^5 report on the algorithmic language Scheme.
R. Kelsey, W. Clinger, J. Rees (editors).
Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.
and ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998.
Available at `http://www.schemers.org/Documents/Standards/`.

[**R6RS**] Revised^6 report on the algorithmic language Scheme.
M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten (editors).
Available at `http://r6rs.org`.

[**R6RSlibraries**] Revised^6 report on the algorithmic language Scheme — Standard Libraries.
M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten (editors).
Available at `http://r6rs.org`.

[**R6RS-Rationale**] Revised^6 report on the algorithmic language Scheme — Rationale.
M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten (editors).
Available at `http://r6rs.org`.

[**R7RS**] Revised^7 report on the algorithmic language Scheme.
A. Shinn, J. Cowan, A. Gleckler (editors).
Available at `http://r7rs.org`.

[**SRFI**] The SRFI web site.
`http://srfi.schemers.org/`

[**SRFI-13**] O. Shivers.
SRFI-13: String libraries.
`http://srfi.schemers.org/srfi-13/`

[**SRFI-130**] J. Cowan.
SRFI-130: Cursor-based string library.
`http://srfi.schemers.org/srfi-130/`

[**DesignNotes**] W. D. Clinger.
Immutable texts.
(This reference consists of rough design notes for the sample implementations. This reference should be removed before the SRFI is finalized.)

[**Unicode**] The Unicode Consortium. Unicode. `http://unicode.org/`

6. Laziness

Generators

[**Red Edition item 12**] What generator library should R7RS-large provide?

> SRFI 121 is a collection of routines for manipulating generators, which are nullary procedures that are invoked repeatedly to generate a sequence of values. They provide lightweight laziness.

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 a series 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.

The names described in this section comprise the `(scheme generator)` library.

Generators can be divided into two classes, finite and infinite. Both kinds of generators can be invoked an indefinite number of times. After a finite generator has generated all its values, it will return an end-of-file object for all subsequent calls. A generator is said to be *exhausted* if calling it will return an end-of-file object. By definition, infinite generators can never be exhausted.

A generator is said to be in an *undefined state* if it cannot be determined exactly how many values it has generated. This arises because it is impossible to tell by inspecting a generator whether it is exhausted or not. For example, `(generator-fold + 0 (generator 1 2 3) (generator 1 2))` will compute $0 + 1 + 1 + 2 + 2 = 6$, at which time the second generator will be exhausted. If the first generator is invoked, however, it may return either 3 or an end-of-file object, depending on whether the implementation of `generator-fold` has invoked it or not. Therefore, the first generator is said to be in an undefined state.

Generator constructors

The result of a generator constructor is just a procedure, so printing it doesn't show much. In the examples in this section we use `generator->list` to convert the generator to a list.

These procedures have names ending with `generator`.

`(generator` *arg …*`)` → *generator* — generator library procedure

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.

`(make-iota-generator` *count start step*`)` → *generator* — generator library procedure
`(make-iota-generator` *count start*`)` → *generator* — generator library procedure
`(make-iota-generator` *count*`)` → *generator* — generator library procedure

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.

```
(generator->list (make-iota-generator 3 8))
   (8 9 10)
```

```
(generator->list (make-iota-generator 3 8 2))
   (8 10 12)
```

`(make-range-generator` *start end step*`)` → *generator* — generator library procedure
`(make-range-generator` *start end*`)` → *generator* — generator library procedure
`(make-range-generator` *start*`)` → *generator* — generator library procedure

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.

```
(generator->list (make-range-generator 3) 4)
   (3 4 5 6)
(generator->list (make-range-generator 3 8))
   (3 4 5 6 7)
(generator->list (make-range-generator 3 8 2))
   (3 5 7)
```

`(make-coroutine-generator` *proc*`)` → *generator* generator library procedure

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` *lis*`)` → *generator* generator library procedure
`(vector>generator` *vec start end*`)` → *generator* generator library procedure
`(vector>generator` *vec start*`)` → *generator* generator library procedure
`(vector>generator` *vec*`)` → *generator* generator library procedure
`(reverse-vector>generator` *vec start end*`)` → *generator* generator library procedure
`(reverse-vector>generator` *vec start*`)` → *generator* generator library procedure
`(reverse-vector>generator` *vec*`)` → *generator* generator library procedure
`(string>generator` *str start end*`)` → *generator* generator library procedure
`(string>generator` *str start*`)` → *generator* generator library procedure
`(string>generator` *str*`)` → *generator* generator library procedure
`(bytevector>generator` *bytevector start end*`)` → *generator* generator library procedure
`(bytevector>generator` *bytevector start*`)` → *generator* generator library procedure
`(bytevector>generator` *bytevector*`)` → *generator* generator library procedure

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*) → *generator* generator library procedure

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*) → generator library procedure

A generator constructor similar to SRFI 1's `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)
```

Generator operations

The following procedures accept one or more generators and return a new generator without consuming any elements from the source generator(s). In general, the result will be a finite generator if the arguments are.

The names of these procedures are prefixed with `g`.

(`gcons*` *item … gen*) → *generator* generator library procedure

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` *gen …*`)` → *generator* generator library procedure

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))
  ()
```

`(gcombine` *proc seed gen* gen_2 *…*`)` → generator library procedure

TODO clarify this

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 v_1 v_2 …seed), where v_1, v_2, …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 gen_n generators is exhausted, at which time all the others are in an undefined state.

`(gfilter` *pred gen*`)` → *generator* generator library procedure
`(gremove` *pred gen*`)` → *generator* generator library procedure

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

`(gtake` *gen k padding*`)` → *generator* generator library procedure
`(gtake` *gen k*`)` → *generator* generator library procedure
`(gdrop` *gen k*`)` → *generator* generator library procedure

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*`)` → *generator* generator library procedure
`(gdrop-while` *pred gen*`)` → *generator* generator library procedure

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 =*`)` → *generator* generator library procedure
`(gdelete` *item gen*`)` → *generator* generator library procedure

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 =*) → *generator* generator library procedure
`(gdelete-neighbor-dups` *gen*) → *generator* generator library procedure

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*) → *generator* generator library procedure

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*) → *generator* generator library procedure

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)
```

Consuming generated values

Unless otherwise noted, these procedures consume all the values available from the generator that is passed to them, and therefore will not return if one or more generator arguments are infinite. They have names prefixed with `generator`.

`(generator->list` *generator k*) → *list* generator library procedure
`(generator->list` *generator*) → *list* generator library procedure

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*) → *list* generator library procedure
`(generator->reverse-list` *generator*) → *list* generator library procedure

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*`)` → *vector* generator library procedure
`(generator->vector` *generator*`)` → *vector* generator library procedure

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*`)` → *integer* generator library procedure

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*`)` → *string* generator library procedure
`(generator->string` *generator*`)` → *string* generator library procedure

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* gen_1 gen_2 *...*`)` → *value* generator library procedure

Works like SRFI 1 `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 gen_n 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 gen* gen_2 *...*`)` → *unspecified* generator library procedure

A generator analogue of `for-each` that consumes generated values using side effects. Repeatedly applies proc on the values yielded by gen, gen_2 ... until any one of the generators is exhausted. The values returned from proc are discarded. The procedure terminates when any of the gen_n generators is exhausted, at which time all the others are in an undefined state. Returns an unspecified value.

`(generator-find` *pred gen*`)` → *value* generator library procedure

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` *pred gen*`)` → *integer* generator library procedure

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

`(generator-any` *pred gen*`)` → *value* generator library procedure

TODO This returns either a value from the generator, or false. However, as a return type, *value* or `#f` looks redundant.

Applies *pred* 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` *pred gen*`)` → *value* generator library procedure

TODO *value* or `#t`?

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 …*`)` → *???* generator library procedure

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)
```

Editor: Arthur Gleckler

Last modified: Wed Jun 10 08:57:14 MST 2015

Lazy sequences

[**Red Edition item 13**] What lazy sequence library should R7RS-large provide?

> SRFI 127 is a variant of SRFI 1 for lazy sequences, which are either proper lists or lists with a generator in the tail. They provide medium-weight laziness.

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.

The names described in this section comprise the `(scheme lseq)` library.

Procedure Index

Here is a short list of the procedures provided by this SRFI.

Constructors	`generator->lseq`	
Predicates	`lseq?`	`lseq=?`

Selectors `lseq-car` `lseq-cdr`
`lseq-first` `lseq-rest` `lseq-ref`
`lseq-take` `lseq-drop`

The whole lazy sequence `lseq-realize` `lseq->generator`
`lseq-length`
`lseq-append` `lseq-zip`

Mapping and filtering `lseq-map` `lseq-for-each`
`lseq-filter` `lseq-remove`

Searching `lseq-find` `lseq-find-tail`
`lseq-any` `lseq-every`
`lseq-index`
`lseq-take-while` `lseq-drop-while`
`lseq-member` `lseq-memq` `lseq-memv`

Specification

The templates given below obey the following conventions for procedure formals:

lseq	A lazy sequence
x, y, a, b	Any value
object, value	Any value
n, i	A natural number (an integer >= 0)
proc	A procedure
pred	A procedure whose return value is treated as a boolean
generator	A procedure with no arguments that returns a sequence of values
=	A boolean procedure taking two arguments

To interpret the examples, pretend that they are executed on a Scheme that prints lazy sequences with the syntax of lists, truncating them when they get too long.

Constructors

Every list constructor procedure is also an lseq constructor procedure. The procedure `generator->lseq` constructs an lseq based on the values of a generator. In order to prepend a value to an lseq, simply use `cons`; to prepend more than one value, use SRFI 1's `cons*`.

`(generator->lseq` *generator*`)` → *lseq* lseq library procedure

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)) => (1 2 3 ...)
```

Predicates

`(lseq?` *x*`)` → *boolean* lseq library procedure

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=? lseq*$_1$ *lseq*$_2$`)` → *boolean* lseq library procedure

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 lseq$_1$ and its second argument is from lseq$_2$.

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.

Selectors

`(lseq-car` *lseq*`)` → *object* lseq library procedure
`(lseq-first` *lseq*`)` → *object* lseq library procedure

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* lseq library procedure
`(lseq-rest` *lseq*`)` → *lseq* lseq library procedure

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.

`(lseq-ref` *lseq i*`)` → *value* lseq library procedure

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* lseq library procedure
`(lseq-drop` *lseq i*`)` → *lseq* lseq library procedure

`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.

The whole lazy sequence

`(lseq-realize` *lseq*`)` → *list* lseq library procedure

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*`)` → *generator* lseq library procedure

Returns a generator which when invoked will return all the elements of lseq, including any that have not yet been realized.

`(lseq-length` *olseq*`)` → *integer* lseq library procedure

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 …*`)` → *lseq* lseq library procedure

Returns an lseq that lazily contains all the elements of all the lseqs in order.

`(lseq-zip` *lseq*$_1$ *lseq*$_2$ *…*`)` → *lseq* lseq library procedure

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))
```

Mapping and filtering

`(lseq-map` *proc lseq*$_1$ *lseq*$_2$ *…*`)` → *lseq* lseq library procedure

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 lseq*$_1$ *lseq*$_2$ *...*`)` → *unspecified* lseq library procedure

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* lseq library procedure
`(lseq-remove` *pred lseq*`)` → *lseq* lseq library procedure

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)
```

Searching

The following procedures all search lseqs for the leftmost element satisfying some criterion.

`(lseq-find` *pred lseq*`)` → *value* lseq library procedure

Return the first element of lseq that satisfies predicate pred, or `#f` if no element does. It cannot reliably be applied to lseqs that include `#f` as an element; use `lseq-find-tail` instead. The predicate is guaranteed to be evaluated on the elements of lseq in sequence order, and only as often as necessary.

```
(lseq-find even? '(3 1 4 1 5 9 2 6)) => 4
```

`(lseq-find-tail` *pred lseq*`)` → *lseq or* `#f` lseq library procedure

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*`)` → *lseq* lseq library procedure

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*`)` → *lseq* lseq library procedure

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* $lseq_1$ $lseq_2$ …`)` → *value* lseq library procedure

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* $lseq_1$ $lseq_2$ …`)` → *value* lseq library procedure

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* $lseq_1$ $lseq_2$ …`)` → *integer or #f* lseq library procedure

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* lseq library procedure
`(lseq-member` *x lseq*`)` → *lseq* lseq library procedure
`(lseq-memq` *x lseq*`)` → *lseq* lseq library procedure
`(lseq-memv` *x lseq*`)` → *lseq* lseq library procedure

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 e_i 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.
```

Streams

[**Red Edition item 14**] What stream library should R7RS-large provide?

SRFI 41 provides classical streams based on delay/force. They provide heavyweight laziness. It specifies two libraries, but will be treated as just one for R7RS purposes.

TODO This SRFI uses a lot of good examples, which certainly help us to understand the library. That said, it doesn't match the style of most of the others, which tend to have few or no extended examples. I wonder if some of them should be moved to an appendix? (Of course, any example appendices must follow ALGOL 60 tradition, and include Runge-Kutta integration :).)

TODO Similarly, all of the material from **Utilities** onward is examples, performance notes, etc. Should it too be reorganized?

TODO This SRFI includes separated type signatures for the procedures. I've integrated the types into the prototypes, as per the other SRFIs, but left them in for checking. They will(?) be removed later.

TODO Terminology here is a bit different from some of the other SRFIs. How much do we want to do in smoothing things out?

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.

TODO Delete? or move to rationale? 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.

TODO Reword 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 (streams))

The names described in this section comprise the `(scheme stream)` library.

~~The (streams primitive)~~ This library provides two mutually-recursive abstract data types: An object of the stream abstract data type is a promise that, when forced, is either stream-null or is an object of type stream-pair. An object of the stream-pair abstract data type contains a stream-car and a stream-cdr, which must be a stream. The essential feature of streams is the systematic suspensions of the recursive promises between the two data types.

TODO The Red Edition document says these are to be one library, so probably all evidence of there being two libraries in this version must be removed.

```
stream
 :: (promise stream-null)
 |  (promise (  stream-pair))

stream-pair
 :: (promise  ) × (promise (  stream))
```

The object stored in the stream-car of a stream-pair is a promise that is forced the first time the stream-car is accessed; its value is cached in case it is needed again. The object may have any type, and different stream elements may have different types. If the stream-car is never accessed, the object stored there is never evaluated. Likewise, the stream-cdr is a promise to return a stream, and is only forced on demand.

This library provides eight operators: constructors for stream-null and stream-pairs, type recognizers for streams and the two kinds of streams, accessors for both fields of a stream-pair, and a lambda that creates procedures that return streams.

Constructors

`(stream-null)` → *stream* — stream library procedure

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* — stream library syntax

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.

Recognizers TODO aka predicates?

`(stream?` *object*`)` → *boolean* — stream library procedure

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)` → *boolean* stream library procedure

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)` → *boolean* stream library procedure

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.

Accessors

`(stream-car stream)` → *value* stream library procedure

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* stream library procedure

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*.

Syntactic form

`(stream-lambda args body)` → *procedure* stream library syntax

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
```

The (streams derived) library

TODO Reword to reflect it's all one lib.

The (streams derived) library provides useful procedures and syntax that depend on the primitives defined above. In the operator templates given below, an ellipsis … indicates zero or more repetitions of the preceding subexpression and square brackets […] indicate optional elements. In the type annotations given below, square brackets […] refer to lists, curly braces {…} refer to streams, and nat refers to exact non-negative integers.

`(define-stream` *name args body*`)` → *unspecified???* stream library 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*`)` → *stream* stream library procedure

[ ] → { }
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*`)` → *stream* stream library procedure
`(port->stream)` → *stream* stream library procedure

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)))))))
```

syntax: (stream *object* ...)
Stream is syntax that takes zero or more *object*s and creates a newly-allocated stream containing in its elements the *object*s, in order. Since stream is syntactic, the *object*s are evaluated when they are accessed, not when the stream is created. If no *object*s 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*`)` → *list* stream library procedure
`(stream->list` *stream*`)` → *list* stream library procedure

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* stream library procedure

{ } ... → { }
Stream-append returns a newly-allocated stream containing in its elements those elements contained in its input *stream*s, in order of input. If any of the input *stream*s is infinite, no elements of any of the succeeding input *stream*s 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* stream library procedure

{{ }} … → { }

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 ...*) → *procedure* stream library procedure

... → { }
Stream-constant takes one or more *object*s and returns a newly-allocated stream containing in its elements the *object*s, repeating the *object*s in succession forever.

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

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

`(stream-drop` *n stream*) → *procedure* stream library 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*) → *stream* stream library procedure

(→ 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 sub-sequences 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*) → *stream* stream library procedure

(→ 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*) → *value* stream library procedure

(× →) × × { } →
Stream-fold applies a binary *proc*edure to *base* and the first element of *stream* to compute a new *base*, then applies the *proc*edure 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 …*`)` → *unspecified???* stream library procedure

(× × …) × { } × { } …
Stream-for-each applies a *proc*edure element-wise to corresponding elements of the input *stream*s for its side-effects; it returns nothing. Stream-for-each stops as soon as any of its input *stream*s is exhausted.

TODO I made the return type in the prototype “unspecified”, the above ¶ should probably match that.

The following procedure displays the contents of a file:

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

`(stream-from` *first step*`)` → *stream* stream library procedure
`(stream-from` *first*`)` → *stream* stream library procedure

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* stream library procedure

(→) × → { }
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 2^{32}, exclusive, the following expression creates a stream of pseudo-random integers between 0 and 2^{32}, 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:

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}}}$$

TODO Check that this matches the original SRFI.

The fractions can be calculated by the stream

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

`(stream-length` *stream*`)` → *integer* stream library procedure

{ } → 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*`)` → *stream???* stream library syntax

TODO I'm not sure just now whether a stream-pair is a stream...so not quite sure on the return type.

Stream-let creates a local scope that binds each *var*iable to the value of its corresponding *expr*ession. It additionally binds *tag* to a procedure which takes the bound *var*iables 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 *var*iables, 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* stream library procedure

(× … →) × { } × { } … → { }

Stream-map applies a *proc*edure element-wise to corresponding elements of the input *stream*s, returning a newly-allocated stream containing elements that are the results of those *proc*edure 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 …*`)` → *???* stream library 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 *expr*ession that is evaluated if the *pattern* matches. There are four types of *pattern*s:

- () — Matches the null stream.
- (pat_0 pat_1 …) — Matches a finite stream with length exactly equal to the number of pattern elements.
- (pat_0 pat_1 … . pat_{rest}) — 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 pat_i 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 *expr*ession 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 *pattern*s 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 *expr*ession. Once the matching pattern is found, the corresponding *expr*ession 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 …*`)` → *???* stream library 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 *var*iables 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*`)` → *stream* stream library procedure
`(stream-range` *first past*`)` → *stream* stream library procedure

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*`)` → *value* stream library procedure

{ } × nat →
Stream-ref returns the *n*th 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* stream library procedure

{ } → { }
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* stream library procedure

(×   →) ×   × { } → { }
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*`)` → *stream* stream library procedure

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*`)` → *stream* stream library procedure

(→ 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*`)` → *stream* stream library procedure

(→) × (→ 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*`)` → *stream ...* stream library procedure

(→ (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* $result_0$ ... $result_{n-1}$

where the returned *seed* is the input *seed* to the next call to the generator and $result_i$ indicates how to produce the next element of the i^{th} 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* stream library procedure

{ } × { } × … → {[…]}
Stream-zip takes one or more input *stream*s and returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input *stream*s. The output stream is as long as the shortest input *stream*, if any of the input *stream*s is finite, or is infinite if all the input *stream*s 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.

Utilities

Streams, being the signature structured data type of functional programming languages, find useful expression in conjunction with higher-order functions. Some of these higher-order functions, and their relationship to streams, are described below.

The identity and constant procedures are frequently useful as the recursive base for maps and folds; **(identity *obj*)** always returns *obj*, and **(const *obj*)** creates a procedure that takes any number of arguments and always returns the same *obj*, no matter its arguments:

(define (identity obj) obj)

(define (const obj) (lambda x obj))

Many of the stream procedures take a unary predicate that accepts an element of a stream and returns a boolean. Procedure **(negate *pred?*)** takes a unary predicate and returns a new unary predicate that, when called, returns the opposite boolean value as the original predicate.

```
(define (negate pred?)
  (lambda (x) (not (pred? x))))
```

Negate is useful for procedures like stream-take-while that take a predicate, allowing them to be used in the opposite direction from which they were written; for instance, with the predicate reversed, stream-take-while becomes stream-take-until. Stream-remove is the opposite of stream-filter:

```
(define-stream (stream-remove pred? strm)
  (stream-filter (negate pred?) strm))
```

A section is a procedure which has been partially applied to some of its arguments; for instance, (double *x*), which returns twice its argument, is a partial application of the multiply operator to the number 2. Sections come in two kinds: left sections partially apply arguments starting from the left, and right sections partially apply arguments starting from the right. Procedure **(lsec *proc* *args* ...)** takes a *proc*edure and some prefix of its arguments and returns a new procedure in which those arguments are partially applied. Procedure **(rsec *proc* *args* ...)** takes a *proc*edure and some reversed suffix of its arguments and returns a new procedure in which those arguments are partially applied.

```
(define (lsec proc . args)
  (lambda x (apply proc (append args x))))

(define (rsec proc . args)
  (lambda x (apply proc (reverse
    (append (reverse args) (reverse x))))))
```

Since most of the stream procedures take a stream as their last (right-most) argument, left sections are particularly useful in conjunction with streams.

(define stream-sum (lsec stream-fold + 0))

Function composition creates a new function by partially applying multiple functions, one after the other. In the simplest case there are only two functions, f and g, composed as ((compose f g) *x*)); the composition can be bound to create a new function, as in (define fg (compose f g)). Procedure **(compose *proc* ...)** takes one or more *proc*edures and returns a new procedure that performs the same action as the individual procedures would if called in succession.

```
(define (compose . fns)
  (let comp ((fns fns))
    (cond
      ((null? fns) 'error)
      ((null? (cdr fns)) (car fns))
      (else
        (lambda args
          (call-with-values
            (lambda ()
              (apply
                (comp (cdr fns))
                args))
            (car fns)))))))
```

Compose works with sections to create succinct but highly expressive procedure definitions. The expression to compute the squares of the integers from 1 to 10 given above at stream-unfold could be written by composing stream-map, stream-take-while, and stream-iterate:

```
((compose
  (lsec stream-map (rsec expt 2))
  (lsec stream-take-while (negate (rsec > 10)))
  (lsec stream-iterate (rsec + 1)))
 1)
```

Examples

The examples below show a few of the myriad ways streams can be exploited, as well as a few ways they can trip the unwary user. All the examples are drawn from published sources; it is instructive to compare the Scheme versions to the originals in other languages.

Infinite streams

As a simple illustration of infinite streams, consider this definition of the natural numbers:

```
(define nats
  (stream-cons 0
    (stream-map add1 nats)))
```

The recursion works because it is offset by one from the initial stream-cons. Another sequence that uses the offset trick is this definition of the fibonacci numbers:

```
(define fibs
  (stream-cons 1
    (stream-cons 1
      (stream-map +
        fibs
        (stream-cdr fibs)))))
```

Yet another sequence that uses the same offset trick is the Hamming numbers, named for the mathematician and computer scientist Richard Hamming, defined as all numbers that have no prime factors greater than 5; in other words, Hamming numbers are all numbers expressible as $2^i \cdot 3^j \cdot 5^k$, where i, j and k are non-negative integers. The Hamming sequence starts with 1 2 3 4 5 6 8 9 10 12 and is computed starting with 1, taking 2, 3 and 5 times all the previous elements with stream-map, then merging sub-streams and eliminating duplicates.

```
(define hamming
  (stream-cons 1
    (stream-unique =
      (stream-merge <
        (stream-map (lsec * 2) hamming)
        (stream-map (lsec * 3) hamming)
        (stream-map (lsec * 5) hamming)))))
```

It is possible to have an infinite stream of infinite streams. Consider the definition of power-table:

```
(define power-table
  (stream-of
    (stream-of (expt m n)
      (m in (stream-from 1)))
      (n in (stream-from 2))))
```

which evaluates to an infinite stream of infinite streams:

```
(stream
  (stream 1 4 9 16 25 ...)
  (stream 1 8 27 64 125 ...)
  (stream 1 16 81 256 625 ...)
  ...)
```

But even though it is impossible to display power-table in its entirety, it is possible to select just part of it:

```
(stream->list 10 (stream-ref power-table 1))
    (1 8 27 64 125 216 343 512 729 1000)
```

This example clearly shows that the elements of a stream are computed lazily, as they are needed; (stream-ref power-table 0) is not computed, even when its successor is displayed, since computing it would enter an infinite loop.

Chris Reade shows how to calculate the stream of prime numbers according to the sieve of Eratosthenes, using a method that eliminates multiples of the sifting base with addition rather than division:

```
(define primes (let ()
  (define-stream (next base mult strm)
    (let ((first (stream-car strm))
          (rest (stream-cdr strm)))
      (cond ((< first mult)
              (stream-cons first
                (next base mult rest)))
            ((< mult first)
              (next base (+ base mult) strm))
            (else (next base
                    (+ base mult) rest)))))
  (define-stream (sift base strm)
    (next base (+ base base) strm))
  (define-stream (sieve strm)
    (let ((first (stream-car strm))>
          (rest (stream-cdr strm)))
      (stream-cons first
        (sieve (sift first rest)))))
  (sieve (stream-from 2))))
```

A final example of infinite streams is a functional pearl from Jeremy Gibbons, David Lester and Richard Bird that enumerates the positive rational numbers without duplicates:

```
(define rats
  (stream-iterate
    (lambda (x)
      (let* ((n (floor x)) (y (- x n)))
        (/ (- n -1 y))))
    1))
```

Backtracking via the stream of successes

Philip Wadler describes the *stream of successes* technique that uses streams to perform backtracking search. The basic idea is that each procedure returns a stream of possible results, so that its caller can decide which result it wants; an empty stream signals failure, and causes backtracking to a previous choice point. The stream of successes technique is useful because the program is written as if to simply enumerate all possible solutions; no backtracking is explicit in the code.

The Eight Queens puzzle, which asks for a placement of eight queens on a chessboard so that none of them attack any other, is an example of a problem that can be solved using the stream of successes technique. The algorithm is to place a queen in the first column of a chessboard; any column is satisfactory. Then a queen is placed in the second column, in any position not held in check by the queen in the first column. Then a queen is placed in the third column, in any position not held in check by the queens in the first two columns. And so on, until all eight queens have been placed. If at any point there is no legal placement for the next queen, backtrack to a different legal position for the previous queens, and try again.

The chessboard is represented as a stream of length m, where there are queens in the first m columns, each position in the stream representing the rank on which the queen appears in that column. For example, stream 4 6 1 5 2 8 3 7 represents the following chessboard:

Included graphics: streams2.jpg

Two queens at column i row j and column m row n check each other if their columns i and m are the same, or if their rows j and n are the same, or if they are on the same diagonal with $i + j = m + n$ or $i - j = m - n$. There is no need to test the columns, because the placement algorithm enforces that they differ, so the check? procedure tests if two queens hold each other in check.

```
(define (check? i j m n)
  (or (= j n)
      (= (+ i j) (+ m n))
      (= (- i j) (- m n))))
```

The algorithm walks through the columns, extending position p by adding a new queen in row n with (stream-append p (stream n)). Safe? tests if it is safe to do so, using the utility procedure stream-and.

```
(define (stream-and strm)
  (let loop ((strm strm))
    (cond ((stream-null? strm) #t)
          ((not (stream-car strm)) #f)
          (else (loop (stream-cdr strm))))))

(define (safe? p n)
  (let* ((len (stream-length p))
         (m (+ len 1)))
    (stream-and
      (stream-of
        (not (check? (car ij) (cadr ij) m n))
          (ij in (stream-zip
                   (stream-range 1 m)
                   p))))))
```

Procedure (queens m) returns all the ways that queens can safely be placed in the first m columns.

```
(define (queens m)
  (if (zero? m)
      (stream (stream))
      (stream-of (stream-append p (stream n))
        (p in (queens (- m 1)))
        (n in (stream-range 1 9))
        (safe? p n))))
```

To see the first solution to the Eight Queens problem, say

(stream->list (stream-car (queens 8)))

To see all 92 solutions, say

```
(stream->list
  (stream-map stream->list
    (queens 8)))
```

There is no explicit backtracking in the code. The stream-of expression in queens returns all possible streams that satisfy safe?; implicit backtracking occurs in the recursive call to queens.

Generators and co-routines

It is possible to model generators and co-routines using streams. Consider the task, due to Carl Hewitt, of determining if two trees have the same sequence of leaves:

(same-fringe? = ‘(1 (2 3)) ’((1 2) 3)) #t

(same-fringe? = ‘(1 2 3) ’(1 (3 2))) #f

The simplest solution is to flatten both trees into lists and compare them element-by-element:

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((pair? (car tree))
          (append (flatten (car tree))
                  (flatten (cdr tree))))
        (else (cons (car tree)
                    (flatten (cdr tree))))))

(define (same-fringe? eql? tree1 tree2)
  (let loop ((t1 (flatten tree1))
             (t2 (flatten tree2)))
    (cond ((and (null? t1) (null? t2)) #t)
          ((or (null? t1) (null? t2)) #f)
          ((not (eql? (car t1) (car t2))) #f)
          (else (loop (cdr t1) (cdr t2))))))
```

That works, but requires time to flatten both trees and space to store the flattened versions; if the trees are large, that can be a lot of time and space, and if the fringes differ, much of that time and space is wasted.

Hewitt used a generator to flatten the trees one element at a time, storing only the current elements of the trees and the machines needed to continue flattening them, so same-fringe? could stop early if the trees differ. Dorai Sitaram presents both the generator solution and a co-routine solution, which both involve tricky calls to call-with-current-continuation and careful coding to keep them synchronized.

An alternate solution flattens the two trees to streams instead of lists, which accomplishes the same savings of time and space, and involves code that looks little different than the list solution presented above:

```
(define-stream (flatten tree)
  (cond ((null? tree) stream-null)
        ((pair? (car tree))
          (stream-append
            (flatten (car tree))
            (flatten (cdr tree))))
        (else (stream-cons
                (car tree)
                (flatten (cdr tree))))))
```

```
(define (same-fringe? eql? tree1 tree2)
  (let loop ((t1 (flatten tree1))
             (t2 (flatten tree2)))
    (cond ((and (stream-null? t1)
                (stream-null? t2)) #t)
          ((or  (stream-null? t1)
                (stream-null? t2)) #f)
          ((not (eql? (stream-car t1)
                      (stream-car t2))) #f)
          (else (loop (stream-cdr t1)
                      (stream-cdr t2))))))
```

Note that streams, a data structure, replace generators or co-routines, which are control structures, providing a fine example of how lazy streams enhance modularity.

A pipeline of procedures

Another way in which streams promote modularity is enabling the use of many small procedures that are easily composed into larger programs, in the style of unix pipelines, where streams are important because they allow a large dataset to be processed one item at a time. Bird and Wadler provide the example of a text formatter. Their example uses right-folds:

```
(define (stream-fold-right f base strm)
  (if (stream-null? strm)
      base
      (f (stream-car strm)
         (stream-fold-right f base
           (stream-cdr strm)))))
```

```
(define (stream-fold-right-one f strm)
  (stream-match strm
  ((x) x)
  ((x . xs)
    (f x (stream-fold-right-one f xs)))))
```

Bird and Wadler define text as a stream of characters, and develop a standard package for operating on text, which they derive mathematically (this assumes the line-separator character is a single #\newline):

```
(define (breakon a)
  (stream-lambda (x xss)
    (if (equal? a x)
        (stream-append (stream (stream)) xss)
        (stream-append
          (stream (stream-append
              (stream x) (stream-car xss)))
          (stream-cdr xss)))))
```

```
(define-stream (lines strm)
  (stream-fold-right
    (breakon #\newline)
    (stream (stream))
    strm))
```

```
(define-stream (words strm)
  (stream-filter stream-pair?
    (stream-fold-right
      (breakon #\space)
      (stream (stream))
      strm)))
```

```
(define-stream (paras strm)
  (stream-filter stream-pair?
    (stream-fold-right
      (breakon stream-null)
      (stream (stream))
      strm)))

(define (insert a)
  (stream-lambda (xs ys)
    (stream-append xs (stream a) ys)))

(define unlines
  (lsec stream-fold-right-one
    (insert #\newline)))

(define unwords
  (lsec stream-fold-right-one
    (insert #\space)))

(define unparas
  (lsec stream-fold-right-one
    (insert stream-null)))
```

These versatile procedures can be composed to count words, lines and paragraphs; the normalize procedure squeezes out multiple spaces and blank lines:

```
(define countlines
  (compose stream-length lines))

(define countwords
  (compose stream-length
           stream-concat
           (lsec stream-map words)
           lines))

(define countparas
  (compose stream-length paras lines))

(define parse
  (compose (lsec stream-map
             (lsec stream-map words))
           paras
           lines))

(define unparse
  (compose unlines
           unparas
           (lsec stream-map
             (lsec stream-map unwords))))

(define normalize (compose unparse parse))
```

More useful than normalization is text-filling, which packs as many words onto each line as will fit.

```
(define (greedy m ws)
  (- (stream-length
       (stream-take-while (rsec <= m)
         (stream-scan
           (lambda (n word)
             (+ n (stream-length word) 1))
           -1
           ws))) 1))
```

```
(define-stream (fill m ws)
  (if (stream-null? ws)
      stream-null
      (let* ((n (greedy m ws))
             (fstline (stream-take n ws))
             (rstwrds (stream-drop n ws)))
        (stream-append
          (stream fstline)
          (fill m rstwrds)))))

(define linewords
  (compose stream-concat
           (lsec stream-map words)))

(define textparas
  (compose (lsec stream-map linewords)
           paras
           lines))

(define (filltext m strm)
  (unparse (stream-map (lsec fill m) (textparas strm))))
```

To display *filename* in lines of *n* characters, say:

```
(stream-for-each display
  (filltext n (file->stream filename)))
```

Though each operator performs only a single task, they can be composed powerfully and expressively. The alternative is to build a single monolithic procedure for each task, which would be harder and involve repetitive code. Streams ensure procedures are called as needed.

Persistent data

Queues are one of the fundamental data structures of computer science. In functional languages, queues are commonly implemented using two lists, with the front half of the queue in one list, where the head of the queue can be accessed easily, and the rear half of the queue in reverse order in another list, where new items can easily be added to the end of a queue. The standard form of such a queue holds that the front list can only be null if the rear list is also null:

(define queue-null (cons ‘() ’())

```
(define (queue-null? obj)
  (and (pair? obj) (null? (car obj))))

(define (queue-check f r)
  (if (null? f)
      (cons (reverse r) '())
      (cons f r)))

(define (queue-snoc q x)
  (queue-check (car q) (cons x (cdr q))))

(define (queue-head q)
  (if (null? (car q))
      (error "empty queue: head")
      (car (car q))))

(define (queue-tail q)
  (if (null? (car q))
      (error "empty-head: tail")
      (queue-check (cdr (car q)) (cdr q))))
```

This queue operates in amortized constant time per operation, with two conses per element, one when it is added to the rear list, and another when the rear list is reversed to become the front list. Queue-snoc and queue-head operate in constant time; queue-tail operates in worst-case linear time when the front list is empty.

Chris Okasaki points out that, if the queue is used persistently, its time-complexity rises from linear to quadratic since each persistent copy of the queue requires its own linear-time access. The problem can be fixed by implementing the front and rear parts of the queue as streams, rather than lists, and rotating one element from rear to front whenever the rear list is larger than the front list:

```
(define queue-null
  (cons stream-null stream-null))

(define (queue-null? x)
  (and (pair? x) (stream-null (car x))))

(define (queue-check f r)
  (if (< (stream-length r) (stream-length f))
      (cons f r)
      (cons (stream-append f (stream-reverse r))
            stream-null)))

(define (queue-snoc q x)
  (queue-check (car q) (stream-cons x (cdr q))))

(define (queue-head q)
  (if (stream-null? (car q))
      (error "empty queue: head")
      (stream-car (car q))))

(define (queue-tail q)
  (if (stream-null? (car q))
      (error "empty queue: tail")
      (queue-check (stream-cdr (car q))
                   (cdr q))))
```

Memoization solves the persistence problem; once a queue element has moved from rear to front, it need never be moved again in subsequent traversals of the queue. Thus, the linear time-complexity to access all elements in the queue, persistently, is restored.

Reducing two passes to one

The final example is a lazy dictionary, where definitions and uses may occur in any order; in particular, uses may precede their corresponding definitions. This is a common problem. Many programming languages allow procedures to be used before they are defined. Macro processors must collect definitions and emit uses of text in order. An assembler needs to know the address that a linker will subsequently give to variables. The usual method is to make two passes over the data, collecting the definitions on the first pass and emitting the uses on the second pass. But Chris Reade shows how streams allow the dictionary to be built lazily, so that only a single pass is needed. Consider a stream of requests:

```
(define requests
  (stream
    '(get 3)
    '(put 1 "a")    ; use follows definition
    '(put 3 "c")    ; use precedes definition
    '(get 1)
    '(get 2)
    '(put 2 "b")    ; use precedes definition
    '(put 4 "d")))  ; unused definition
```

We want a procedure that will display cab, which is the result of (get 3), (get 1), and (get 2), in order. We first separate the request stream into gets and puts:

```
(define (get? obj) (eq? (car obj) 'get))

(define-stream (gets strm)
  (stream-map cadr (stream-filter get? strm)))

(define-stream (puts strm)
  (stream-map cdr  (stream-remove get? strm)))
```

Now, run-dict inserts each element of the puts stream into a lazy dictionary, represented as a stream of key/value pairs (an association stream), then looks up each element of the gets stream with stream-assoc:

```
(define-stream (run-dict requests)
  (let ((dict (build-dict (puts requests))))
    (stream-map (rsec stream-assoc dict)
      (gets requests))))

(define (stream-assoc key dict)
    (cond ((stream-null? dict) #f)
          ((equal? key (car (stream-car dict)))
            (stream-car dict))
          (else (stream-assoc key
                  (stream-cdr dict)))))
```

Dict is created in the let, but nothing is initially added to it. Each time stream-assoc performs a lookup, enough of dict is built to satisfy the lookup, but no more. We are assuming that each item is defined once and only once. All that is left is to define the procedure that inserts new items into the dictionary, lazily:

```
(define-stream (build-dict puts)
  (if (stream-null? puts)
      stream-null
      (stream-cons
        (stream-car puts)
        (build-dict (stream-cdr puts)))))
```

Now we can run the requests and print the result:

```
(stream-for-each display
  (stream-map cadr (run-dict requests)))
```

The (put 4 “d”) definition is never added to the dictionary because it is never needed.

Pitfalls

Programming with streams, or any lazy evaluator, can be tricky, even for programmers experienced in the genre. Programming with streams is even worse in Scheme than in a purely functional language, because, though the streams are lazy, the surrounding Scheme expressions in which they are embedded are eager. The impedance between lazy and eager can occasionally lead to astonishing results. Thirty-two years ago, William Burge warned:

For example, a simple version of stream-map that returns a stream built by applying a unary procedure to the elements of an input stream could be defined like this:

```
(define-stream (stream-map proc strm) ;wrong!
  (let loop ((strm strm))
    (if (stream-null? strm)
        stream-null
        (stream-cons
          (proc (stream-car strm))
          (loop (stream-cdr strm))))))
```

That looks right. It properly wraps the procedure in stream-lambda, and the two legs of the if both return streams, so it type-checks. But it fails because the named let binds loop to a procedure using normal lambda rather than stream-lambda, so even though the first element of the result stream is lazy, subsequent elements are eager. Stream-map can be written using stream-let:

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

Here, stream-let assures that each element of the result stream is properly delayed, because each is subject to the stream-lambda that is implicit in stream-let, so the result is truly a stream, not a "list in disguise." Another version of this procedure was given previously at the description of define-stream.

Another common problem occurs when a stream-valued procedure requires the next stream element in its definition. Consider this definition of stream-unique:

```
(define-stream (stream-unique eql? strm) ;wrong!
  (stream-match strm
    (() strm)
    ((_) strm)
    ((a b . _)
      (if (eql? a b)
          (stream-unique eql?
            (stream-cdr strm))
          (stream-cons a
            (stream-unique eql?
              (stream-cdr strm)))))))
```

The (a b . _) pattern requires the value of the next stream element after the one being considered. Thus, to compute the n^{th} element of the stream, one must know the $n{+}1^{\text{st}}$ element, and to compute the $n{+}1^{\text{st}}$ element, one must know the $n{+}2^{\text{nd}}$ element, and to compute.... The correct version, given above in the description of stream-drop-while, only needs the current stream element.

A similar problem occurs when the stream expression uses the previous element to compute the current element:

```
(define (nat n)
  (stream-ref
    (stream-let loop ((s (stream 0)))
      (stream-cons (stream-car s)
        (loop (stream (add1 (stream-car s))))))
    n))
```

This program traverses the stream of natural numbers, building the stream as it goes. The definition is correct; (nat 15) evaluates to 15. But it needlessly uses unbounded space because each stream element holds the value of the prior stream element in the binding to s.

When traversing a stream, it is easy to write the expression in such a way that evaluation requires unbounded space, even when that is not strictly necessary. During the discussion of SRFI-40, Joe Marshall created this infamous procedure:

```
(define (times3 n)
  (stream-ref
    (stream-filter
      (lambda (x)
        (zero? (modulo x n)))
      (stream-from 0))
    3))
```

(times3 5) evaluates to 15 and (times3 #e1e9) evaluates to three billion, though it takes a while. In either case, times3 should operate in bounded space, since each iteration mutates the promise that holds the next value. But it is easy to write times3 so that it does not operate in bounded space, as the follies of SRFI-40 showed. The common problem is that some element of the stream (often the first element) is bound outside the expression that is computing the stream, so it holds the head of the stream, which holds the second element, and so on. In addition to testing the programmer, this procedure tests the stream primitives (it caught several errors during development) and also tests the underlying Scheme system (it found a bug in one implementation).

Laziness is no defense against an infinite loop; for instance, the expression below never returns, because the odd? predicate never finds an odd stream element.

```
(stream-null?
  (stream-filter odd?
    (stream-from 0 2)))
```

Ultimately, streams are defined as promises, which are implemented as thunks (lambda with no arguments). Since a stream is a procedure, comparisons such as eq?, eqv? and equal? are not meaningful when applied to streams. For instance, the expression (define s ((stream-lambda () stream-null))) defines s as the null stream, and (stream-null? s) is #t, but (eq? s stream-null) is #f. To determine if two streams are equal, it is necessary to evaluate the elements in their common prefixes, reporting #f if two elements ever differ and #t if both streams are exhausted at the same time.

```
(define (stream-equal? eql? xs ys)
  (cond ((and (stream-null? xs)
              (stream-null? ys)) #t)
        ((or (stream-null? xs)
             (stream-null? ys)) #f)
        ((not (eql? (stream-car xs)
                    (stream-car ys))) #f)
        (else (stream-equal? eql?
                (stream-cdr xs)
                (stream-cdr ys)))))
```

It is generally not a good idea to mix lazy streams with eager side-effects, because the order in which stream elements are evaluated determines the order in which the side-effects occur. For a simple example, consider this side-effecting version of strm123:

```
(define strm123-with-side-effects
  (stream-cons (begin (display "one") 1)
    (stream-cons (begin (display "two") 2)
      (stream-cons (begin (display "three") 3)
        stream-null))))
```

The stream has elements 1 2 3. But depending on the order in which stream elements are accessed, “one”, “two” and “three” could be printed in any order.

Since the performance of streams can be very poor, normal (eager) lists should be preferred to streams unless there is some compelling reason to the contrary. For instance, computing pythagorean triples with streams

```
(stream-ref
  (stream-of (list a b c)
    (n in (stream-from 1))
    (a in (stream-range 1 n))
    (b in (stream-range a n))
    (c is (- n a b))
    (= (+ (* a a) (* b b)) (* c c)))
  50)
```

is about two orders of magnitude slower than the equivalent expression using loops. TODO Explain why...what features of an implementation can make streams perform better.

```
(do ((n 1 (+ n 1))) ((> n 228))
  (do ((a 1 (+ a 1))) ((> a n))
    (do ((b a (+ b 1))) ((> b n))
      (let ((c (- n a b)))
        (if (= (+ (* a a) (* b b)) (* c c))
            (display (list a b c)))))))
```

References

Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts. Second edition, 1996. mitpress.mit.edu/sicp. The classic text on computer science. Section 3.5 includes extensive discussion of odd streams.

Anne L. Bewig. "Golden Ratio" (personal communication). Homework for the high school course *Calculus.* Teaching my daughter how to calculate the 200th element of a continued fraction was a moment of sheer joy in the development of the stream libraries.

Philip L. Bewig. *Scheme Request for Implementation 40: A Library of Streams.* August, 2004. srfi.schemers.org/srfi-40. Describes an implementation of the stream data type.

Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988. The classic text on functional programming. Even streams are discussed in the context of purely functional programming.

William H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975. An early text on functional programming, and still one of the best, though the terminology is dated. Discusses even streams in Section 3.10.

Jeremy Gibbons, David Lester and Richard Bird, "Functional Pearl: Enumerating the Rationals," under consideration for publication in *Journal of Functional Programming.* `http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/rationals.pdf`. Discusses a series of expressions that enumerate the rational numbers without duplicates.

Carl Hewitt. "Viewing control structures as patterns of passing messages," in *Journal of Artificial Intelligence*, Volume 8, Number 3 (June, 1977), pp 323-364. Also published as Artificial Intelligence Memo 410 by the Massachusetts Institute of Technology, `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-410.pdf`. Describes the Actor message-passing system; one of the examples used is the same-fringe? problem.

Peter J. Landin. "A correspondence between ALGOL 60 and Church's lambda-notation: Part I," *Communications of the ACM*, Volume 8, Number 2, February 1965., pages 89–101. The seminal description of streams.

Joe Marshall. "Stream problem redux", from *Usenet comp.lang.scheme*, June 28, 2002. groups.google.com/group/comp.lang.scheme/m
The original post on comp.lang.scheme that describes the times3 problem.

Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 2003. Revised version of Okasaki's thesis *Purely Functional Data Structures*, Carnegie-Mellon University, 1996, www.cs.cmu.edu/~rwh/theses/okasaki.pdf. Provides a strong defense of laziness, and describes several data structures that exploit laziness, including streams and queues.

Stephen K. Park and Keith W. Miller. "Random number generators: good ones are hard to find," *Communications of the ACM*, Volume 31, Issue 10 (October 1988), pages 1192–1201. Describes a minimal standard random number generator.

Simon Peyton-Jones, et al, editors. *Haskell 98: Haskell 98 Language and Libraries: The Revised Report.* December 2002. www.haskell.org/onlinereport. Haskell is the prototypical purely functional language, and includes even streams, which it calls lists, as its fundamental structured data type.

Chris Reade. *Elements of Functional Programming.* Addison-Wesley, April 1989. A textbook on functional programming.

Antoine de Saint-Exupéry. Chapter III "L'Avion" of *Terre des Hommes.* 1939. "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Dorai Sitaram. *Teach Yourself Scheme in Fixnum Days.* www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html. A useful introduction to Scheme; includes generator and co-routine solutions to the same-fringe? problem.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton von Straaten, editors. *Revised6 Report on the Algorithmic Language Scheme.* September 26, 2007. www.r6rs.org. The standard definition of the Scheme programming language.

André van Tonder. *Scheme Request for Implementation 45: Primitives for Expressing Iterative Lazy Algorithms.* srfi.schemers.org/srfi-45. April, 2004. Describes the problems inherent in the promise data type of R5RS (also present in R6RS), and provides the alternate promise data type used in the stream primitives.

Philip Wadler. "How to replace failure by a list of successes," in *Proceedings of the conference on functional programming languages and computer architecture*, Nancy, France, 1985, pages 113–128. Describes the "list of successes" technique for implementing backtracking algorithms using streams.

Philip Wadler, Walid Taha, and David MacQueen, "How to add laziness to a strict language without even being odd." 1998 ACM SIGPLAN Workshop on ML, pp. 24ff. homepages.inf.ed.ac.uk/wadler/papers/lazyinstrict/lazyinstrict.ps. Describes odd and even styles of lazy evaluation, and shows how to add lazy evaluation to the strict functional language SML.

All cited web pages visited during September 2007.

Editor: Michael Sperber

Miscellaneous

Boxes

[**Red Edition item 15**] What box library should R7RS-large provide? *

SRFI 111 is a trivial library for boxes (single-slot records).

A box is a container for an object of any Scheme type, including another box. It is like a single-element vector, or half of a pair, or a direct representation of state. Boxes are normally used as minimal mutable storage, and can inject a controlled amount of mutability into an otherwise immutable data structure (or one that is conventionally treated as immutable). They can be used to implement call-by-reference semantics by passing a boxed value to a procedure and expecting the procedure to mutate the box before returning.

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

The names described in this section comprise the `(scheme box)` library.

Procedures

The following procedures implement the box type (which is disjoint from all other Scheme types).~~, and are exported by the (srfi 111) library (or (srfi :111) on R6RS).~~

`(box` *value*`)` → *box* box library procedure

Constructor. Returns a newly allocated box initialized to *value*.

`((box?` *object*`)` → *boolean* box library procedure

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

`(unbox` *box*`)` → *value* box library procedure

Accessor. Returns the current value of *box*.

`(set-box!` *box value*`)` → *unspecified?* box library procedure

Mutator. Changes *box* to hold *value*.

The behavior of boxes with the equivalence predicates `eq?`, `eqv?`, and `equal?` is the same as if they were implemented with records. That is, two boxes are both `eq?` and `eqv?` iff they are the product of the same call to `box` and not otherwise, and while they must be `equal?` if they are `eqv?`, the converse is implementation-dependent.

Autoboxing (optional)

The following provisions of this SRFI are optional: TODO Are they optional in R7RS-Large?

- A procedure, whether system-provided or user-written, that expects a box as an argument but receives a non-box may, if appropriate, allocate a box itself that holds the value, thus providing autoboxing.
- A procedure that accepts arguments only of specified types (such as `+`) but receives a box instead may, if appropriate, unbox the box. Procedures that accept arguments of any type (such as `cons`) must not unbox their arguments.
- Calling `unbox` on a non-box may simply return the non-box.

Editor: Mike Sperber

Last modified: Wed Jul 3 09:04:14 MST 2013

List queues

[**Red Edition item 16**] What list queue library should R7RS-large provide?

> SRFI 117 provides mutable half-deques with $O(1)$ addition at both ends and $O(1)$ deletion from the front. They are based directly on ordinary Scheme lists, and are known in other Lisps as “tconc structures”.

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.

The names described in this section comprise the `(scheme list-queue)` library.

Constructors

`(make-list-queue` *list last*`)` → *list-queue* list-queue library procedure
`(make-list-queue` *list*`)` → *list-queue* list-queue library procedure

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 …*`)` → list-queue library list-queue

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*`)` → *list-queue* list-queue library procedure

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*`)` → *list-queue* list-queue library procedure
`(list-queue-unfold` *stop? mapper successor seed*`)` → list-queue library list-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*`)` → list-queue library list-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.

Predicates

`(list-queue?` *obj*`)` → *boolean* list-queue library procedure

Returns `#t` if obj is a list queue, and `#f` otherwise. This operation is $O(1)$.

`(list-queue-empty?` *list-queue*`)` → *boolean* list-queue library procedure

Returns `#t` if list-queue has no elements, and `#f` otherwise. This operation is $O(1)$.

Accessors

`(list-queue-front` *list-queue*`)` → *v* list-queue library procedure
alue

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*`)` → *value* list-queue library procedure

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*`)` → *list* list-queue library procedure

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*`)` → *value value* list-queue library procedure

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 $O(1)$.

Mutators

`(list-queue-add-front!` *list-queue element*`)` → *unspecified* list-queue library procedure

Adds element to the beginning of list-queue. Returns an unspecified value. This operation is $O(1)$.

`(list-queue-add-back!` *list-queue element*`)` → *unspecified* list-queue library procedure

Adds element to the end of list-queue. Returns an unspecified value. This operation is $O(1)$.

`(list-queue-remove-front!` *list-queue*`)` → *value* list-queue library procedure

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*`)` → *value* list-queue library procedure

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*`)` → *list* list-queue library procedure

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*`)` → *unspecified* list-queue library procedure
`(list-queue-set-list!` *list-queue list*`)` → *unspecified* list-queue library procedure

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 ...*`)` → *list-queue* list-queue library procedure

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.

TODO Does this belong in a subsection entitled “Mutators”?

`(list-queue-append!` *list-queue ...*`)` → *list-queue* list-queue library procedure

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*`)` → *list-queue* list-queue library procedure

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`.

Mapping

`(list-queue-map` *proc list-queue*`)` → *list-queue* list-queue library procedure

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*`)` → *list-queue* list-queue library procedure

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*`)` → *unspecified* list-queue library procedure

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.

Editor: Michael Sperber

Last modified: Tue Jun 9 08:44:01 MST 2015

Ephemerons

[**Red Edition item 17**] What ephemeron library should R7RS-large provide?

> SRFI 124 provides ephemerons compatible with MIT Scheme except that they are immutable. A dummy implementation for systems without GC support is also provided.

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.

Much of this specification is derived with thanks from the MIT Scheme Reference Manual.

The names described in this section comprise the `(scheme ephemeron)` library.

Specification

`(ephemeron?` *object*`)` → *boolean* ephemeron library procedure

Returns `#t` if *object* is an ephemeron; otherwise returns `#f`.

`(make-ephemeron` *key datum*`)` → *ephemeron* ephemeron library procedure

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*`)` → *boolean* ephemeron library procedure

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*`)` → *value* ephemeron library procedure
`(ephemeron-datum` *ephemeron*`)` → *value* ephemeron library procedure

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*`)` → *???* ephemeron library procedure

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.

TODO What does "optional" mean here? Can a conforming implementation just not export it? Or always generate an error? Is there a cond-expand feature for this?

References

The original paper on ephemerons is Barry Hayes, "Ephemerons: a New Finalization Mechanism", *Object-Oriented Languages, Programming, Systems, and Applications*, 1997.

A useful implementation paper is Bruno Haible, "Weak References: Data Types and Implementation" posted 2005-04-24.

"Eliminating Cycles in Weak Tables" is about the addition of ephemeron tables to Lua 5.2. It explains how Lua's tricolor mark-sweep GC is extended to handle them, unfortunately using the O(N^2) algorithm of Hayes 1997.

"Extending Garbage Collection to Complex Data Structures" extends ephemerons, which have a fixed GC strategy, to a new data structure called *blobs* that explain to the GC how they are to be processed, thus allowing an arbitrary number of key-like and value-like pointers.

Verse

TODO With all respect to gls, does this stay in the final report?

Reclaimer, spare that tree!
Take not a single bit!
It used to point to me,
Now I'm protecting it.
It was the reader's `cons`
That made it, paired by dot;
Now, GC, for the nonce,
Thou shalt reclaim it not.

That old familiar tree,
Whose `cdr`s and whose `car`s
Are spread o'er memory —
And wouldst thou it unparse?
GC, cease and desist!
In it no free list store;

Oh spare that moby list
Now pointing throughout core!

It was my parent tree
When it was circular;
It pointed then to me:
I was its `cadadr`.
My `cdr` was a list,
My `car` a dotted pair —
That tree will sore be missed
If it remains not there.

And now I to thee point,
A saving root, old friend!
Thou shalt remain disjoint
From free lists to the end.
Old tree! The sweep still brave!
And, GC, mark this well:
While I exist to save,
Thou shan't reclaim one cell.

The Great Quux (with apologies to George Pope Morris)

(Pedantic note: In Scheme, pairs constructed by `read` are officially immutable, so this scenario is technically impossible.)

Editor: Arthur A. Gleckler

Titlecase library

[**Red Edition item 18**] The vote was for no titlecase library.

This section is left in as a placeholder, and will be removed later in the editing process.

Comparators

[**Red Edition item 19**] What comparator library should R7RS-large provide?

> SRFI 128 is a comparator library whose interface is used by SRFI 113 and SRFI 125. It was not formally balloted but was voted in based on the votes for those libraries, as `(scheme comparator)`.

Post-finalization note: Because of the extremely high cost of conforming to the first and third conditions of `default-hash`, implementers may disregard those conditions and examine only a bounded portion of the argument.

This SRFI provides *comparators*, which bundle a type test predicate, an equality predicate, an ordering predicate, and a hash function (the last two are optional) into a single Scheme object. By packaging these procedures together, they can be treated as a single item for use in the implementation of data structures.

The names described in this section comprise the `(scheme comparator)` library.

~~The procedures in this SRFI are in the `(srfi 128)` library (or `(srfi :128)` on R6RS), but the sample implementation currently places them in the `(comparators)` library. This means it can't be used alongside SRFI 114, but there's no reason for anyone to do that.~~

Definitions

A *comparator* is an object of a disjoint type. It is a bundle of procedures that are useful for comparing two objects in a total order. It is an error if any of the procedures have side effects. 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 *ordering predicate* returns `#t` if the first object precedes the second in a total order, and `#f` otherwise. Note that if it is true, the equality predicate must be false. It is the programmer's responsibility to ensure that it is irreflexive, antisymmetric, transitive, and can handle any arguments that satisfy the type test predicate.
- The *hash function* takes an object and returns an exact non-negative integer. It is the programmer's responsibility to ensure that it can handle any argument that satisfies the type test predicate, and that it returns the same value on two objects if the equality predicate says they are the same (but not necessarily the converse).

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

Limitations

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

Index

- Predicates:
  `comparator ?comparator-ordered? comparator-hashable?`
- Constructors:
  `make-comparator make-pair-comparator make-list-comparator`
  `make-vector-comparator make-eq-comparator make-eqv-comparator`
  `make-equal-comparator`
- Standard hash functions:
  `boolean-hash char-hash char-ci-hash string-hash`
  `string-ci-hash symbol-hash number-hash`
- Bounds and salt:
  `hash-bound hash-salt`
- Default comparators:
  `make-default-comparator default-hash comparator-register-default!`
- Accessors and invokers:
  `comparator-type-test-predicate comparator-equality-predicate`
  `comparator-ordering-predicate comparator-hash-function`
  `comparator-test-type comparator-check-type comparator-hash`
- Comparison predicates:
  `=? <? >? <=? >=?`
- Syntax: `comparator-if<=>`

Predicates

`(comparator? obj)` → *boolean* comparator library procedure

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

`(comparator-ordered? comparator)` → *boolean* comparator library procedure

Returns `#t` if *comparator* has a supplied ordering predicate, and `#f` otherwise.

`(comparator-hashable? comparator)` → *boolean* comparator library procedure

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

Constructors

The following comparator constructors all supply appropriate type test predicates, equality predicates, ordering predicates, and hash functions based on the supplied arguments. They are allowed to cache their results: they need not return a newly allocated object, since comparators are pure and functional. In addition, the procedures in a comparator are likewise pure and functional.

`(make-comparator type-test equality ordering hash)` → *comparator* comparator library procedure

Returns a comparator which bundles the *type-test, equality, ordering*, and *hash* procedures provided. However, if *ordering* or *hash* is `#f`, a procedure is provided that signals an error on application. The predicates `comparator-ordered?` and/or `comparator-hashable?`, respectively, will return `#f` in these cases.

Here are calls on `make-comparator` that will return useful comparators for standard Scheme types:

- `(make-comparator boolean? boolean=? (lambda (x y) (and (not x) y)) boolean-hash)` will return a comparator for booleans, expressing the ordering `#f` < `#t` and the standard hash function for booleans.
- `(make-comparator real? = < (lambda (x) (exact (abs x))))` will return a comparator expressing the natural ordering of real numbers and a plausible (but not optimal) hash function.
- `(make-comparator string? string=? string<? string-hash)` will return a comparator expressing the implementation's ordering of strings and the standard hash function.
- `(make-comparator string? string-ci=? string-ci<? string-ci-hash` will return a comparator expressing the implementation's case-insensitive ordering of strings and the standard case-insensitive hash function.

`(make-pair-comparator car-comparator cdr-comparator)` → *comparator* comparator library procedure

This procedure returns comparators whose functions behave as follows. TODO why plural?

- The type test returns `#t` if its argument is a pair, if the car satisfies the type test predicate of car-comparator, and the cdr satisfies the type test predicate of cdr-comparator.
- The equality function returns `#t` if the cars are equal according to car-comparator and the cdrs are equal according to cdr-comparator, and `#f` otherwise.
- The ordering function first compares the cars of its pairs using the equality predicate of car-comparator. If they are equal, then the ordering predicate of car-comparator is applied to the cars and its value is returned. Otherwise, the predicate compares the cdrs using the equality predicate of cdr-comparator. If they are not equal, then the ordering predicate of cdr-comparator is applied to the cdrs and its value is returned.

- The hash function computes the hash values of the car and the cdr using the hash functions of car-comparator and cdr-comparator respectively and then hashes them together in an implementation-defined way.

`(make-list-comparator` *element-comparator type-test empty? head tail*`)` → *comparator* comparator library procedure

This procedure returns comparators whose functions behave as follows: TODO plural?

- The type test returns `#t` if its argument satisfies type-test and the elements satisfy the type test predicate of element-comparator.
- The total order defined by the equality and ordering functions is as follows (known as lexicographic order):
  - The empty sequence, as determined by calling empty?, compares equal to itself.
  - The empty sequence compares less than any non-empty sequence.
  - Two non-empty sequences are compared by calling the head procedure on each. If the heads are not equal when compared using element-comparator, the result is the result of that comparison. Otherwise, the results of calling the tail procedure are compared recursively.
- The hash function computes the hash values of the elements using the hash function of element-comparator and then hashes them together in an implementation-defined way.

`(make-vector-comparator` *element-comparator type-test length ref*`)` → *comparator* comparator library procedure

This procedure returns comparators whose functions behave as follows: TODO plural?

- The type test returns `#t` if its argument satisfies type-test and the elements satisfy the type test predicate of element-comparator.
- The equality predicate returns `#t` if both of the following tests are satisfied in order: the lengths of the vectors are the same in the sense of =, and the elements of the vectors are the same in the sense of the equality predicate of element-comparator.
- The ordering predicate returns `#t` if the results of applying length to the first vector is less than the result of applying length to the second vector. If the lengths are equal, then the elements are examined pairwise using the ordering predicate of element-comparator. If any pair of elements returns `#t`, then that is the result of the list comparator's ordering predicate; otherwise the result is `#f`
- The hash function computes the hash values of the elements using the hash function of element-comparator and then hashes them together in an implementation-defined way.

Here is an example, which returns a comparator for byte vectors:

```
(make-vector-comparator
  (make-comparator exact-integer? = < number-hash)
  bytevector?
  bytevector-length
  bytevector-u8-ref)
```

`(make-eq-comparator)` → *comparator* comparator library procedure
`(make-eqv-comparator)` → *comparator* comparator library procedure
`(make-equal-comparator)` → *comparator* comparator library procedure

These procedures return comparators whose functions behave as follows:

- The type test returns `#t` in all cases.

- The equality functions are `eq?`, `eqv?`, and `equal?` respectively.
- The ordering function is implementation-defined, except that it must conform to the rules for ordering functions. It may signal an error instead.
- The hash function is `default-hash`.

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

Standard hash functions

These are hash functions for some standard Scheme types, suitable for passing to `make-comparator`. Users may write their own hash functions with the same signature. However, if programmers wish their hash functions to be backward compatible with the reference implementation of SRFI 69, they are advised to write their hash functions to accept a second argument and ignore it.

`(boolean-hash`*obj*`)` → *integer*	comparator library procedure
`(char-hash`*obj*`)` → *integer*	comparator library procedure
`(char-ci-hash`*obj*`)` → *integer*	comparator library procedure
`(string-hash`*obj*`)` → *integer*	comparator library procedure
`(string-ci-hash`*obj*`)` → *integer*	comparator library procedure
`(symbol-hash`*obj*`)` → *integer*	comparator library procedure
`(number-hash`*obj*`)` → *integer*	comparator library procedure

These are suitable hash functions for the specified types. The hash functions `char-ci-hash` and `string-ci-hash` treat their argument case-insensitively. Note that while `symbol-hash` may return the hashed value of applying `symbol->string` and then `string-hash` to the symbol, this is not a requirement.

Bounds and salt

The following macros allow the callers of hash functions to affect their behavior without interfering with the calling signature of a hash function, which accepts a single argument (the object to be hashed) and returns its hash value. They are provided as macros so that they may be implemented in different ways: as a global variable, a SRFI 39 or R7RS parameter, or an ordinary procedure, whatever is most efficient in a particular implementation.

`(hash-bound)` → *integer* comparator library syntax

Hash functions should be written so as to return a number between 0 and the largest reasonable number of elements (such as hash buckets) a data structure in the implementation might have. What that value is depends on the implementation. This value provides the current bound as a positive exact integer, typically for use by user-written hash functions. However, they are not required to bound their results in this way.

`(hash-salt)` → *integer* comparator library syntax

A salt is random data in the form of a non-negative exact integer used as an additional input to a hash function in order to defend against dictionary attacks, or (when used in hash tables) against denial-of-service attacks that overcrowd certain hash buckets, increasing the amortized O(1) lookup time to O(n). Salt can also be used to specify which of a family of hash functions should be used for purposes such as cuckoo hashing. This macro provides the current value of the salt, typically for use by user-written hash functions. However, they are not required to make use of the current salt.

The initial value is implementation-dependent, but must be less than the value of `(hash-bound)`, and should be distinct for distinct runs of a program unless otherwise specified by the implementation. Implementations may provide a means to specify the salt value to be used by a particular invocation of a hash function.

Default comparators

`(make-default-comparator)` → *integer* comparator library procedure

Returns a comparator known as a default comparator that accepts Scheme values and orders them in some implementation-defined way, subject to the following conditions:

- 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 both type *a* and type *b* compare equal to each other. This is not permitted for any of the Scheme types mentioned below.
- The empty list must be ordered before all pairs.
- When comparing booleans, it must use the total order `#f` < `#t`.
- When comparing characters, it must use `char=?` and `char<?`.

  Note: In R5RS, this is an implementation-dependent order that is typically the same as Unicode codepoint order; in R6RS and R7RS, it is Unicode codepoint order.
- When comparing pairs, it must behave the same as a comparator returned by `make-pair-comparator` with default comparators as arguments.
- When comparing symbols, it must use an implementation-dependent total order. One possibility is to use the order obtained by applying `symbol->string` to the symbols and comparing them using the total order implied by `string<?`.
- When comparing bytevectors, it must behave the same as a comparator created by the expression `(make-vector-comparator (make-comparator < = number-hash) bytevector? bytevector-length bytevector-u8-ref)`.
- When comparing numbers where either number is complex, 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 somewhat surprising results if one real part is exact and the other is inexact.
- When comparing real numbers, it must use `=` and `<`.
- When comparing strings, it must use `string=?` and `string<?`.

  Note: In R5RS, this is lexicographic order on the implementation-dependent order defined by `char<?`; in R6RS it is lexicographic order on Unicode codepoint order; in R7RS it is an implementation-defined order.
- When comparing vectors, it must behave the same as a comparator returned by `(make-vector-comparator (make-default-comparator) vector? vector-length vector-ref)`.
- When comparing members of types registered with `comparator-register-default!`, it must behave in the same way as the comparator registered using that function.

Default comparators use `default-hash` as their hash function.

`(default-hash` *obj*`)` → *value* comparator library procedure

This is the hash function used by default comparators, which accepts a Scheme value and hashes it in some implementation-defined way, subject to the following conditions:

- When applied to a pair, it must return the result of hashing together the values returned by `default-hash` when applied to the car and the cdr.

- When applied to a boolean, character, string, symbol, or number, it must return the same result as `boolean-hash`, `char-hash`, `string-hash`, `symbol-hash`, or `number-hash` respectively.
- When applied to a list or vector, it must return the result of hashing together the values returned by `default-hash` when applied to each of the elements.

`comparator-register-default!` comparatorprocedureunspecified

Registers comparator for use by default comparators, such that if the objects being compared both satisfy the type test predicate of comparator, it will be employed by default comparators to compare them. Returns an unspecified value. It is an error if any value satisfies both the type test predicate of comparator and any of the following type test predicates: `boolean?`, `char?`, `null?`, `pair?`, `symbol?`, `bytevector?`, `number?`, `string?`, `vector?`, or the type test predicate of a comparator that has already been registered.

This procedure is intended only to extend default comparators into territory that would otherwise be undefined, not to override their existing behavior. In general, the ordering of calls to `comparator-register-default!` should be irrelevant. However, implementations that support inheritance of record types may wish to ensure that default comparators always check subtypes before supertypes.

Accessors and Invokers

`(comparator-type-test-predicate` *comparator*`)` → *procedure* comparator library procedure
`(comparator-equality-predicate` *comparator*`)` → *procedure* comparator library procedure
`(comparator-ordering-predicate` *comparator*`)` → *procedure* comparator library procedure
`(comparator-hash-function` *comparator*`)` → *procedure* comparator library procedure

Return the four procedures of *comparator*.

`(comparator-test-type` *comparator obj*`)` → *boolean* comparator library procedure

Invokes the type test predicate of *comparator* on *obj* and returns what it returns. More convenient than `comparator-type-test-predicate`, but less efficient when the predicate is called repeatedly.

`(comparator-check-type` *comparator obj*`)` → *boolean* comparator library procedure

Invokes the type test predicate of *comparator* on *obj* and returns true if it returns true, but signals an error otherwise. More convenient than `comparator-type-test-predicate`, but less efficient when the predicate is called repeatedly.

TODO the return type here is misleading; if it's a boolean, then it's true. We don't have a standard way, in a prototype of saying "may signal an error". I'm not sure what to do here; my instinct is just to leave it as is.

`(comparator-hash` *comparator obj*`)` → *integer* comparator library procedure

Invokes the hash function of *comparator* on obj and returns what it returns. More convenient than `comparator-hash-function`, but less efficient when the function is called repeatedly.

Note: No invokers are required for the equality and ordering predicates, because `=?` and `<?` serve this function.

Comparison predicates

`(=?` *comparator* $object_1$ $object_2$ $object_3$ *...*`)` → *boolean* comparator library procedure
`(<?` *comparator* $object_1$ $object_2$ $object_3$ *...*`)` → *boolean* comparator library procedure
`(>?` *comparator* $object_1$ $object_2$ $object_3$ *...*`)` → *boolean* comparator library procedure

`(<=?` *comparator object*$_1$ *object*$_2$ *object*$_3$ *...*`)` → *boolean* comparator library procedure
`(>=?` *comparator object*$_1$ *object*$_2$ *object*$_3$ *...*`)` → *boolean* comparator library procedure

These procedures are analogous to the number, character, and string comparison predicates of Scheme. They allow the convenient use of comparators to handle variable data types.

These procedures apply the equality and ordering predicates of *comparator* to the *objects* as follows. If the specified relation returns `#t` for all *object*$_i$ and *object*$_j$ where n is the number of objects and $1 <= i < j <= n$, then the procedures return `#t`, but otherwise `#f`. Because the relations are transitive, it suffices to compare each object with its successor. The order in which the values are compared is unspecified.

Syntax

`(comparator-if<=>` *<comparator> <object*$_1$*> <object*$_2$*> <less-than> <equal-to> <greater-than>*`)` → *value*
comparator library syntax
`(comparator-if<=>` *<object*$_1$*> <object*$_2$*> <less-than> <equal-to> <greater-than>*`)` → *value*
comparator library syntax

It is an error unless <comparator> evaluates to a comparator and <object$_1$> and <object$_2$> evaluate to objects that the comparator can handle. If the ordering predicate returns true when applied to the values of <object$_1$> and <object$_2$> in that order, then <less-than> is evaluated and its value returned. If the equality predicate returns true when applied in the same way, then <equal-to> is evaluated and its value returned. If neither returns true, <greater-than> is evaluated and its value returned.

If <comparator> is omitted, a default comparator is used.

TODO remove angle brackets, use std param notation

Editor: Arthur A. Gleckler