Revised$^{5.92}$ Report on the Algorithmic Language Scheme
— Standard Libraries —

Michael Sperber
William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten
(*Editors*)
Richard Kelsey, Jonathan Rees
(*Editors, Revised^5 Report on the Algorithmic Language Scheme*)
18 January 2007

SUMMARY

The report gives a defining description of the standard libraries of the programming language Scheme.
Chapter 1 describes the library implementing Unicode semantics for characters and strings, chapter 2 describes the library for handling binary data, chapter 3 describes the library containing list utilities procedures, chapter 5 describes the record system, 6 describes the libraries for exceptions and conditions, chapter 7 describes the I/O libraries, chapter 9 describes specialized libraries for dealing with numbers and arithmetic, chapter 10 describes the `syntax-case` facility for writing arbitrary macros, chapter 11 describes the library for hash tables, chapter 12 describes the enumerations library, and chapter 13 describes various miscellaneous libraries.
Chapter 14 describes the composite library containing most of the forms described in this report. Chapter 15 describes the `eval` facility for evaluating Scheme expressions represented as data. Chapter 16 describes the operations for mutating pairs. Chapter 17 describes a library with some procedures from the previous version of this report for backwards compatibility.
The report concludes with a list of references and an alphabetic index.
This report frequently refers back to the *Revised^6 Report on the Algorithmic Language Scheme* [9]; references to the report are identified by designations such as "report section" or "report chapter".

*** DRAFT***

This is a preliminary draft. It is intended to reflect the decisions taken by the editors' committee, but contains many mistakes, ambiguities and inconsistencies.

CONTENTS

1. Unicode

The procedures exported by the `(r6rs unicode)` library provide access to some aspects of the Unicode semantics for characters and strings: category information, case-independent comparisons, case mappings, and normalization [10].

Some of the procedures that operate on characters or strings ignore the difference between upper case and lower case. The procedures that ignore case have "`-ci`" (for "case insensitive") embedded in their names.

1.1. Characters

`(char-upcase` *char*`)` procedure
`(char-downcase` *char*`)` procedure
`(char-titlecase` *char*`)` procedure
`(char-foldcase` *char*`)` procedure

These procedures take a character argument and return a character result. If the argument is an upper case or title case character, and if there is a single character that is its lower case form, then `char-downcase` returns that character. If the argument is a lower case or title case character, and there is a single character that is its upper case form, then `char-upcase` returns that character. If the argument is a lower case or upper case character, and there is a single character that is its title case form, then `char-titlecase` returns that character. Finally, if the character has a case-folded character, then `char-foldcase` returns that character. Otherwise the character returned is the same as the argument. For Turkic characters İ (`#\x130`) and ı (`#\x131`), `char-foldcase` behaves as the identity function; otherwise `char-foldcase` is the same as `char-downcase` composed with `char-upcase`.

```
(char-upcase #\i)          ⟹ #\I
(char-downcase #\i)        ⟹ #\i
(char-titlecase #\i)       ⟹ #\I
(char-foldcase #\i)        ⟹ #\i

(char-upcase #\ß)          ⟹ #\ß
(char-downcase #\ß)        ⟹ #\ß
(char-titlecase #\ß)       ⟹ #\ß
(char-foldcase #\ß)        ⟹ #\ß

(char-upcase #\Σ)          ⟹ #\Σ
(char-downcase #\Σ)        ⟹ #\σ
(char-titlecase #\Σ)       ⟹ #\Σ
(char-foldcase #\Σ)        ⟹ #\σ

(char-upcase #\ς)          ⟹ #\Σ
(char-downcase #\ς)        ⟹ #\ς
(char-titlecase #\ς)       ⟹ #\Σ
(char-foldcase #\ς)        ⟹ #\σ
```

Note: These procedures are consistent with Unicode's locale-independent mappings from scalar values to scalar values for upcase, downcase, titlecase, and case-folding operations. These mappings can be extracted from `UnicodeData.txt` and `CaseFolding.txt` from the Unicode Consortium, ignoring Turkic mappings in the latter.

Note that these character-based procedures are an incomplete approximation to case conversion, even ignoring the user's locale. In general, case mappings require the context of a string, both in arguments and in result. The `string-upcase`, `string-downcase`, `string-titlecase`, and `string-foldcase` procedures (section 1.2) perform more general case conversion.

`(char-ci=?` $char_1$ $char_2$ $char_3$ ...`)` procedure
`(char-ci<?` $char_1$ $char_2$ $char_3$ ...`)` procedure
`(char-ci>?` $char_1$ $char_2$ $char_3$ ...`)` procedure
`(char-ci<=?` $char_1$ $char_2$ $char_3$ ...`)` procedure
`(char-ci>=?` $char_1$ $char_2$ $char_3$ ...`)` procedure

These procedures are similar to `char=?` et cetera, but operate on the case-folded versions of the characters.

```
(char-ci<? #\z #\Z)        ⟹ #f
(char-ci=? #\z #\Z)        ⟹ #t
(char-ci=? #\ς #\σ)        ⟹ #t
```

`(char-alphabetic?` *char*`)` procedure
`(char-numeric?` *char*`)` procedure
`(char-whitespace?` *char*`)` procedure
`(char-upper-case?` *letter*`)` procedure
`(char-lower-case?` *letter*`)` procedure
`(char-title-case?` *letter*`)` procedure

These procedures return `#t` if their arguments are alphabetic, numeric, whitespace, upper case, lower case, or title case characters, respectively; otherwise they return `#f`.

A character is alphabetic if it is a Unicode letter, i.e. if it is in one of the categories Lu, Ll, Lt, Lm, and Lo. A character is numeric if it is in category Nd. A character is whitespace if it is in one of the space, line, or paragraph separator categories (Zs, Zl or Zp), or if is U+0009 (Horizontal tabulation), U+000A (Line feed), U+000B (Vertical tabulation), U+000C (Form feed), or U+000D (Carriage return). A character is upper case if it has the Unicode "Uppercase" property, lower case if it has the "Lowercase" property, and title case if it is in the Lt general category.

```
(char-alphabetic? #\a)     ⟹ #t
(char-numeric? #\1)        ⟹ #t
(char-whitespace? #\space) ⟹ #t
(char-whitespace? #\x00A0) ⟹ #t
(char-upper-case? #\Σ)     ⟹ #t
(char-lower-case? #\σ)     ⟹ #t
(char-lower-case? #\x00AA) ⟹ #t
(char-title-case? #\I)     ⟹ #f
```

```
(char-title-case? #\x01C5)  ⟹ #t
```

`(char-general-category` *char*`)` procedure

Returns a symbol representing the Unicode general category of *char*, one of `Lu`, `Ll`, `Lt`, `Lm`, `Lo`, `Mn`, `Mc`, `Me`, `Nd`, `Nl`, `No`, `Ps`, `Pe`, `Pi`, `Pf`, `Pd`, `Pc`, `Po`, `Sc`, `Sm`, `Sk`, `So`, `Zs`, `Zp`, `Zl`, `Cc`, `Cf`, `Cs`, `Co`, or `Cn`.

```
(char-general-category #\a) ⟹ Ll
(char-general-category #\space)
          ⟹ Zs
(char-general-category #\x10FFFF)
          ⟹ Cn
```

1.2. Strings

`(string-upcase` *string*`)` procedure
`(string-downcase` *string*`)` procedure
`(string-titlecase` *string*`)` procedure
`(string-foldcase` *string*`)` procedure

These procedures take a string argument and return a string result. They are defined in terms of Unicode's locale-independent case mappings from Unicode-scalar-value sequences to scalar-value sequences. In particular, the length of the result string can be different from the length of the input string. When the specified result is equal in the sense of `string=?` to the argument, these procedures may return the argument instead of a newly allocated string.

The `string-upcase` procedure converts a string to upper case; `string-downcase` converts a string to lowercase. The `string-foldcase` procedure converts the string to its case-folded counterpart, using the full case-folding mapping, but without the special mappings for Turkic languages. The `string-titlecase` procedure converts the first character to title case in each contiguous sequence of cased characters within *string*, and it downcases all other cased characters; for the purposes of detecting cased-character sequences, case-ignorable characters are ignored (i.e., they do not interrupt the sequence).

```
(string-upcase "Hi")        ⟹ "HI"
(string-downcase "Hi")      ⟹ "hi"
(string-foldcase "Hi")      ⟹ "hi"

(string-upcase "Straße")    ⟹ "STRASSE"
(string-downcase "Straße")  ⟹ "straße"
(string-foldcase "Straße")  ⟹ "strasse"
(string-downcase "STRASSE") ⟹ "strasse"

(string-downcase "Σ")       ⟹ "σ"

; Chi Alpha Omicron Sigma:
(string-upcase "ΧΑΟΣ")      ⟹ "ΧΑΟΣ"
(string-downcase "ΧΑΟΣ")    ⟹ "χαος"
(string-downcase "ΧΑΟΣΣ")   ⟹ "χαοσς"
(string-downcase "ΧΑΟΣ Σ")  ⟹ "χαος σ"
(string-foldcase "ΧΑΟΣΣ")   ⟹ "χαοσσ"
(string-upcase "χαος")      ⟹ "ΧΑΟΣ"
(string-upcase "χαοσ")      ⟹ "ΧΑΟΣ"

(string-titlecase "kNock KNoCK")
                            ⟹ "Knock Knock"
(string-titlecase "who's there?")
                            ⟹ "Who's There?"
(string-titlecase "r6rs")   ⟹ "R6Rs"
(string-titlecase "R6RS")   ⟹ "R6Rs"
```

Note: The case mappings needed for implementing these procedures can be extracted from `UnicodeData.txt`, `SpecialCasing.txt`, `WordBreakProperty.txt` (the "MidLetter" property partly defines case-ignorable characters), and `CaseFolding.txt` from the Unicode Consortium.

Since these procedures are locale-independent, they may not be appropriate for some locales.

`(string-ci=?` $string_1$ $string_2$ $string_3$ ...`)` procedure
`(string-ci<?` $string_1$ $string_2$ $string_3$ ...`)` procedure
`(string-ci>?` $string_1$ $string_2$ $string_3$ ...`)` procedure
`(string-ci<=?` $string_1$ $string_2$ $string_3$ ...`)` procedure
`(string-ci>=?` $string_1$ $string_2$ $string_3$ ...`)` procedure

These procedures are similar to `string=?` et cetera, but operate on the case-folded versions of the strings.

```
(string-ci<? "z" "Z")       ⟹ #f
(string-ci=? "z" "Z")       ⟹ #t
(string-ci=? "Straße" "Strasse")
                            ⟹ #t
(string-ci=? "Straße" "STRASSE")
                            ⟹ #t
(string-ci=? "ΧΑΟΣ" "χαοσ")
                            ⟹ #t
```

`(string-normalize-nfd` *string*`)` procedure
`(string-normalize-nfkd` *string*`)` procedure
`(string-normalize-nfc` *string*`)` procedure
`(string-normalize-nfkc` *string*`)` procedure

These procedures take a string argument and return a string result, which is the input string normalized to Unicode normalization form D, KD, C, or KC, respectively. When the specified result is equal in the sense of `string=?` to the argument, these procedures may return the argument instead of a newly allocated string.

```
(string-normalize-nfd "\xE9;")
                            ⟹ "\x65;\x301;"
(string-normalize-nfc "\xE9;")
                            ⟹ "\xE9;"
```

```
(string-normalize-nfd "\x65;\x301;")
                              ⟹ "\x65;\x301;"
(string-normalize-nfc "\x65;\x301;")
                              ⟹ "\xE9;"
```

2. Bytevectors

Many applications must deal with blocks of binary data by accessing them in various ways—extracting signed or unsigned numbers of various sizes. Therefore, the `(r6rs bytevector)` library provides a single type for blocks of binary data with multiple ways to access that data. It deals only with integers in various sizes with specified endianness, because these are the most frequent applications.

Bytevectors are objects of a disjoint type. Conceptually, a bytevector represents a sequence of 8-bit bytes. The description of bytevectors uses the term *byte* for an exact integer in the interval $\{-128, \ldots, 127\}$ and the term *octet* for an exact integer in the interval $\{0, \ldots, 255\}$. A byte corresponds to its two's complement representation as an octet.

The length of a bytevector is the number of bytes it contains. This number is fixed. A valid index into a bytevector is an exact, non-negative integer. The first byte of a bytevector has index 0; the last byte has an index one less than the length of the bytevector.

Generally, the access procedures come in different flavors according to the size of the represented integer and the endianness of the representation. The procedures also distinguish signed and unsigned representations. The signed representations all use two's complement.

Like list and vector literals, literals representing bytevectors must be quoted:

```
'#vu8(12 23 123)              ⟹ #vu8(12 23 123)
```

2.1. General operations

`(endianness` ⟨endianness symbol⟩`)` syntax

⟨Endianness symbol⟩ must be a symbol describing an endianness. An implementation must support at least the symbols `big` and `little`, but may support other endianness symbols. `(endianness` ⟨endianness symbol⟩`)` evaluates to the symbol named ⟨endianness symbol⟩. Whenever one of the procedures operating on bytevectors accepts an endianness as an argument, that argument must be one of these symbols. It is a syntax violation for ⟨endianness symbol⟩ to be anything other than an endianness symbol supported by the implementation.

Note: Implementors are encouraged to use widely accepted designations for endianness symbols other than `big` and `little`.

`(native-endianness)` procedure

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

`(bytevector?` *obj*`)` procedure

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

`(make-bytevector` *k*`)` procedure
`(make-bytevector` *k fill*`)` procedure

Returns a newly allocated bytevector of *k* bytes.

If the *fill* argument is missing, the initial contents of the returned bytevector are unspecified.

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

`(bytevector-length` *bytevector*`)` procedure

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

`(bytevector=?` $bytevector_1$ $bytevector_2$`)` procedure

Returns `#t` if $bytevector_1$ and $bytevector_2$ are equal—that is, if they have the same length and equal bytes at all valid indices. It returns `#f` otherwise.

`(bytevector-fill!` *bytevector fill*`)`

The *fill* argument is as in the description of the `make-bytevector` procedure. Stores *fill* in every element of *vector* and returns the unspecified value. Analogous to `vector-fill!`.

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

Source-start, *target-start*, and *k* must be non-negative exact integers that satisfy

$$\begin{array}{rcccccl} 0 & \leq & \textit{source-start} & \leq & \textit{source-start} + k & \leq & l_{source} \\ 0 & \leq & \textit{target-start} & \leq & \textit{target-start} + k & \leq & l_{target} \end{array}$$

where l_{source} is the length of *source* and l_{target} is the length of *target*.

The `bytevector-copy!` procedure copies the bytes from *source* at indices

$$\{source\text{-}start, \ldots source\text{-}start + k - 1\}$$

to consecutive indices in *target* starting at *target-index*.

This must work even if the memory regions for the source and the target overlap, i.e., the bytes at the target location after the copy must be equal to the bytes at the source location before the copy.

This returns the unspecified value.

```
(let ((b (u8-list->bytevector '(1 2 3 4 5 6 7 8))))
  (bytevector-copy! b 0 b 3 4)
  (bytevector->u8-list b))  ⟹ (1 2 3 1 2 3 4 8)
```

`(bytevector-copy` *bytevector*`)` procedure

Returns a newly allocated copy of *bytevector*.

2.2. Operations on bytes and octets

`(bytevector-u8-ref` *bytevector k*`)` procedure
`(bytevector-s8-ref` *bytevector k*`)` procedure

K must be a valid index of *bytevector*.

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

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

```
(let ((b1 (make-bytevector 16 -127))
      (b2 (make-bytevector 16 255)))
  (list
    (bytevector-s8-ref b1 0)
    (bytevector-u8-ref b1 0)
    (bytevector-s8-ref b2 0)
    (bytevector-u8-ref b2 0))⟹ (-127 129 -1 255)
```

`(bytevector-u8-set!` *bytevector k octet*`)` procedure
`(bytevector-s8-set!` *bytevector k byte*`)` procedure

K must be a valid index of *bytevector*.

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

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

Both procedures return the unspecified value.

```
(let ((b (make-bytevector 16 -127)))

  (bytevector-s8-set! b 0 -126)
  (bytevector-u8-set! b 1 246)

  (list
    (bytevector-s8-ref b 0)
    (bytevector-u8-ref b 0)
    (bytevector-s8-ref b 1)
    (bytevector-u8-ref b 1))⟹ (-126 130 -10 246)
```

`(bytevector->u8-list` *bytevector*`)` procedure
`(u8-list->bytevector` *list*`)` procedure

List must be a list of octets.

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

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

`(bytevector-uint-ref` *bytevector k endianness size*`)` procedure
`(bytevector-sint-ref` *bytevector k endianness size*`)` procedure
`(bytevector-uint-set!` *bytevector k n endianness size*`)` procedure
`(bytevector-sint-set!` *bytevector k n endianness size*`)` procedure

Size must be a positive exact integer. $\{k, \ldots, k + size - 1\}$ must be valid indices of *bytevector*.

`bytevector-uint-ref` retrieves the exact integer corresponding to the unsigned representation of size *size* and specified by *endianness* at indices $\{k, \ldots, k + size - 1\}$.

`bytevector-sint-ref` retrieves the exact integer corresponding to the two's complement representation of size *size* and specified by *endianness* at indices $\{k, \ldots, k + size - 1\}$.

For `bytevector-uint-set!`, *n* must be an exact integer in the set $\{0, \ldots, 256^{size} - 1\}$.

`bytevector-uint-set!` stores the unsigned representation of size *size* and specified by *endianness* into *bytevector* at indices $\{k, \ldots, k + size - 1\}$.

For `bytevector-sint-set!`, *n* must be an exact integer in the interval $\{-256^{size}/2, \ldots, 256^{size}/2 - 1\}$. `bytevector-sint-set!` stores the two's complement representation of size *size* and specified by *endianness* into *bytevector* at indices $\{k, \ldots, k + size - 1\}$.

The `...-set!` procedures return the unspecified value.

```
(define b (make-bytevector 16 -127))

(bytevector-uint-set! b 0 (- (expt 2 128) 3)
                      (endianness little) 16)

(bytevector-uint-ref b 0 (endianness little) 16)
         ⟹
    #xfffffffffffffffffffffffffffffffd

(bytevector-sint-ref b 0 (endianness little) 16)
         ⟹ -3

(bytevector->u8-list b)
         ⟹ (253 255 255 255 255 255 255 255
             255 255 255 255 255 255 255 255)

(bytevector-uint-set! b 0 (- (expt 2 128) 3)
                (endianness big) 16)

(bytevector-uint-ref b 0 (endianness big) 16)
         ⟹
    #xfffffffffffffffffffffffffffffffd

(bytevector-sint-ref b 0 (endianness big) 16)
         ⟹ -3

(bytevector->u8-list b)
         ⟹ (255 255 255 255 255 255 255 255
             255 255 255 255 255 255 255 253))
```

2.3. Operations on integers of arbitrary size

`(bytevector->uint-list` *bytevector endianness size*`)` procedure
`(bytevector->sint-list` *bytevector endianness size*`)` procedure
`(uint-list->bytevector` *list endianness size*`)` procedure
`(sint-list->bytevector` *list endianness size*`)` procedure

Size must be a positive exact integer.

These procedures convert between lists of integers and their consecutive representations according to *size* and *endianness* in the *bytevector* objects in the same way as `bytevector->u8-list` and `u8-list->bytevector` do for one-byte representations.

```
(let ((b (u8-list->bytevector '(1 2 3 255 1 2 1 2))))
  (bytevector->sint-list b (endianness little) 2))
         ⟹ (513 -253 513 513)

(let ((b (u8-list->bytevector '(1 2 3 255 1 2 1 2))))
  (bytevector->uint-list b (endianness little) 2))
         ⟹ (513 65283 513 513)
```

2.4. Operations on 16-bit integers

`(bytevector-u16-ref` *bytevector k endianness*`)` procedure
`(bytevector-s16-ref` *bytevector k endianness*`)` procedure
`(bytevector-u16-native-ref` *bytevector k*`)` procedure
`(bytevector-s16-native-ref` *bytevector k*`)` procedure
`(bytevector-u16-set!` *bytevector k n endianness*`)` procedure
`(bytevector-s16-set!` *bytevector k n endianness*`)` procedure
`(bytevector-u16-native-set!` *bytevector k n*`)` procedure
`(bytevector-s16-native-set!` *bytevector k n*`)` procedure

K must be a valid index of *bytevector*; so must $k + 1$.

These retrieve and set two-byte representations of numbers at indices k and $k + 1$, according to the endianness specified by *endianness*. The procedures with `u16` in their names deal with the unsigned representation; those with `s16` in their names deal with the two's complement representation.

The procedures with `native` in their names employ the native endianness, and work only at aligned indices: k must be a multiple of 2.

The `...-set!` procedures return the unspecified value.

```
(define b
  (u8-list->bytevector
    '(255 255 255 255 255 255 255 255
      255 255 255 255 255 255 255 253)))

(bytevector-u16-ref b 14 (endianness little))
         ⟹ 65023
(bytevector-s16-ref b 14 (endianness little))
         ⟹ -513
(bytevector-u16-ref b 14 (endianness big))
         ⟹ 65533
(bytevector-s16-ref b 14 (endianness big))
         ⟹ -3

(bytevector-u16-set! b 0 12345 (endianness little))
(bytevector-u16-ref b 0 (endianness little))
         ⟹ 12345

(bytevector-u16-native-set! b 0 12345)
(bytevector-u16-native-ref b 0) 12345

(bytevector-u16-ref b 0 (endianness little))
         ⟹ unspecified
```

2.5. Operations on 32-bit integers

`(bytevector-u32-ref` *bytevector k endianness*`)` procedure
`(bytevector-s32-ref` *bytevector k endianness*`)` procedure
`(bytevector-u32-native-ref` *bytevector k*`)` procedure
`(bytevector-s32-native-ref` *bytevector k*`)` procedure
`(bytevector-u32-set!` *bytevector k n endianness*`)` procedure
`(bytevector-s32-set!` *bytevector k n endianness*`)` procedure
`(bytevector-u32-native-set!` *bytevector k n*`)` procedure
`(bytevector-s32-native-set!` *bytevector k n*`)` procedure

$\{k, \ldots, k+3\}$ must be valid indices of *bytevector*..

These retrieve and set four-byte representations of numbers at indices $\{k, \ldots, k+3\}$, according to the endianness specified by *endianness*. The procedures with `u32` in their names deal with the unsigned representation, those with `s32` with the two's complement representation.

The procedures with `native` in their names employ the native endianness, and work only at aligned indices: k must be a multiple of 4.

The `...-set!` procedures return the unspecified value.

```
(define b
  (u8-list->bytevector
    '(255 255 255 255 255 255 255 255
      255 255 255 255 255 255 255 253)))

(bytevector-u32-ref b 12 (endianness little))
          ⟹ 4261412863
(bytevector-s32-ref b 12 (endianness little))
          ⟹ -33554433
(bytevector-u32-ref b 12 (endianness big))
          ⟹ 4294967293
(bytevector-s32-ref b 12 (endianness big))
          ⟹ -3
```

`(bytevector-u64-ref` *bytevector k endianness*`)` procedure
`(bytevector-s64-ref` *bytevector k endianness*`)` procedure
`(bytevector-u64-native-ref` *bytevector k*`)` procedure
`(bytevector-s64-native-ref` *bytevector k*`)` procedure
`(bytevector-u64-set!` *bytevector k n endianness*`)` procedure
`(bytevector-s64-set!` *bytevector k n endianness*`)` procedure
`(bytevector-u64-native-set!` *bytevector k n*`)` procedure
`(bytevector-s64-native-set!` *bytevector k n*`)` procedure

$\{k, \ldots, k+7\}$ must be valid indices of *bytevector*.

These retrieve and set eight-byte representations of numbers at indices $\{k, \ldots, k+7\}$, according to the endianness specified by *endianness*. The procedures with `u64` in their names deal with the unsigned representation, those with `s64` with the two's complement representation.

The procedures with `native` in their names employ the native endianness, and work only at aligned indices: k must be a multiple of 8.

The `...-set!` procedures return the unspecified value.

```
(define b
  (u8-list->bytevector
    '(255 255 255 255 255 255 255 255
      255 255 255 255 255 255 255 253)))

(bytevector-u64-ref b 8 (endianness little))
          ⟹ 18302628885633695743
(bytevector-s64-ref b 8 (endianness little))
          ⟹ -144115188075855873
(bytevector-u64-ref b 8 (endianness big))
          ⟹ 18446744073709551613
(bytevector-s64-ref b 8 (endianness big))
          ⟹ -3
```

`(bytevector-ieee-single-native-ref` *bytevector k*`)` procedure
`(bytevector-ieee-single-ref` *bytevector k endianness*`)` procedure

$\{k, \ldots, k+3\}$ must be valid indices of *bytevector*. For `bytevector-ieee-single-native-ref`, k must be a multiple of 4.

These procedures return the inexact real that best represents the IEEE-754 single precision number represented by the four bytes beginning at index k.

2.6. Operations on IEEE-754 numbers

`(bytevector-ieee-double-native-ref` *bytevector k*`)` procedure
`(bytevector-ieee-double-ref` *bytevector k endianness*`)` procedure

$\{k, \ldots, k+7\}$ must be valid indices of *bytevector*. For `bytevector-ieee-double-native-ref`, k must be a multiple of 8.

These procedures return the inexact real that best represents the IEEE-754 single precision number represented by the eight bytes beginning at index k.

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

$\{k, \ldots, k+3\}$ must be valid indices of *bytevector*. For `bytevector-ieee-single-native-set!`, k must be a multiple of 4. X must be a real number.

These procedures store an IEEE-754 single precision representation of x into elements k through $k+3$ of *bytevector*, and returns the unspecified value.

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

$\{k, \ldots, k+7\}$ must be valid indices of *bytevector*. For `bytevector-ieee-double-native-set!`, k must be a multiple of 8.

These procedures store an IEEE-754 double precision representation of x into elements k through $k+7$ of *bytevector*, and returns the unspecified value.

3. List utilities

This chapter describes the `(r6rs lists)` library.

`(find` *proc list*`)` procedure

Proc must be a procedure that takes a single argument and returns a single value. *Proc* must not mutate *list*. The `find` procedure applies *proc* to the elements of *list* in order. If *proc* returns a true value for an element, `find` immediately returns that element. If *proc* returns `#f` for all elements of the list, it returns `#f`. *Proc* is always called in same dynamic environment as `find` itself.

```
(find even? '(3 1 4 1 5 9)) ⟹ 4
(find even? '(3 1 5 1 5 9)) ⟹ #f
```

Implementation responsibilities: The implementation must check that *list* is a chain of pairs up to the found element, or that it is indeed a list if no element is found. It should not check that it is a chain of pairs beyond the found element. The implementation must check the restrictions on *proc* to the extent performed by applying it as described.

`(for-all` *proc* $list_1$ $list_2$ ... $list_n$`)` procedure
`(exists` *proc* $list_1$ $list_2$ ... $list_n$`)` procedure

The *list*s must all have the same length, and *proc* must be a procedure that accepts n arguments and returns a single value. *Proc* must not mutate the *list* arguments.

For natural numbers $i = 0, 1, \ldots$, the `for-all` procedure successively applies *proc* to arguments $x_i^1 \ldots x_i^n$, where x_i^j is the ith element of $list_j$, until `#f` is returned. If *proc* returns true values for all but the last element of $list_1$, *for-all* performs a tail call of *proc* on the kth elements, where k is the length of $list_1$. If *proc* returns `#f` on any set of elements, `for-all` returns `#f` after the first such application of *proc*. If the *list*s are all empty, `for-all` returns `#t`.

For natural numbers $i = 0, 1, \ldots$, the `exists` procedure applies *proc* successively to arguments $x_i^1 \ldots x_i^n$, where x_i^j is the ith element of $list_j$, until a true value is returned. If *proc* returns `#f` for all but the last elements of the *list*s, *exists* performs a tail call of *proc* on the kth elements, where k is the length of $list_1$—in this case, the *list*s must all be lists of length k. If *proc* returns a true value on any set of elements, `exists` returns that value after the first such application of *proc*. If the *list*s are all empty, `exists` returns `#f`.

Proc is always called in same dynamic environment as `for-all` or, respectively, `exists` itself.

```
(for-all even? '(3 1 4 1 5 9))
          ⟹ #f
(for-all even? '(3 1 4 1 5 9 . 2))
          ⟹ #f
(for-all even? '(2 4 14))   ⟹ #t
(for-all even? '(2 4 14 . 9))
          ⟹ &assertion exception
(for-all (lambda (n) (and (even? n) n)) '(2 4 14))
          ⟹ 14
(for-all < '(1 2 3) '(2 3 4))⟹ #t
(for-all < '(1 2 4) '(2 3 4))⟹ #f

(exists even? '(3 1 4 1 5 9))
          ⟹ #t
(exists even? '(3 1 1 5 9)) ⟹ #f
(exists even? '(3 1 1 5 9 . 2))
          ⟹ &assertion exception
(exists (lambda (n) (and (even? n) n)) '(2 1 4 14))
          ⟹ 2
(exists < '(1 2 4) '(2 3 4))⟹ #t
(exists > '(1 2 3) '(2 3 4))⟹ #f
```

Implementation responsibilities: The implementation must check that the *list*s are chains of pairs to the extent necessary to determine the return value. If this requires traversing the lists entirely, the implementation should check that the *list*s all have the same length. If not, it should not check that the *list*s are chains of pairs beyond the traversal. The implementation must check the restrictions on *proc* to the extent performed by applying it as described.

`(filter` *proc list*`)` procedure
`(partition` *proc list*`)` procedure

Proc must be a procedure that accepts a single argument and returns a single value. *Proc* must not mutate *list*. The

`filter` procedure successively applies *proc* to the elements of *list* and returns a list of the values of *list* for which *proc* returned a true value. The `partition` procedure also successively applies *proc* to the elements of *list*, but returns two values, the first one a list of the values of *list* for which *proc* returned a true value, and the second a list of the values of *list* for which *proc* returned `#f`. *Proc* is always called in same dynamic environment as `filter` or, respectively, `partition` itself.

```
(filter even? '(3 1 4 1 5 9 2 6))
          ⟹ (4 2 6)

(partition even? '(3 1 4 1 5 9 2 6))
          ⟹ (4 2 6) (3 1 1 5 9) ; two values
```

Implementation responsibilities: The implementation must check the restrictions on *proc* to the extent performed by applying it as described.

`(fold-left` *combine nil list*$_1$ *list*$_2$ ... *list*$_n$`)` procedure

The *list*s must all have the same length. *Combine* must be a procedure that takes one more argument than there are *list*s and returns a single value. *Combine* must not mutate the *list* arguments. The `fold-left` procedure iterates the *combine* procedure over an accumulator value and the values of the *list*s from left to right, starting with an accumulator value of *nil*. More specifically, `fold-left` returns *nil* if the *list*s are empty. If they are not empty, *combine* is first applied to *nil* and the respective first elements of the *list*s in order. The result becomes the new accumulator value, and *combine* is applied to new accumulator value and the respective next elements of the *list*. This step is repeated until the end of the list is reached; then the accumulator value is returned. *Combine* is always called in same dynamic environment as `fold-left` itself.

```
(fold-left + 0 '(1 2 3 4 5))⟹ 15

(fold-left (lambda (a e) (cons e a)) '()
           '(1 2 3 4 5))
          ⟹ (5 4 3 2 1)

(fold-left (lambda (count x)
             (if (odd? x) (+ count 1) count))
           0
           '(3 1 4 1 5 9 2 6 5 3))
          ⟹ 7

(fold-left (lambda (max-len s)
             (max max-len (string-length s)))
           0
           '("longest" "long" "longer"))
          ⟹ 7

(fold-left cons '(q) '(a b c))
          ⟹ ((((q) . a) . b) . c)

(fold-left + 0 '(1 2 3) '(4 5 6))
          ⟹ 21
```

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.

`(fold-right` *combine nil list*$_1$ *list*$_2$ ... *list*$_n$`)` procedure

The *list*s must all have the same length. *Combine* must be a procedure that takes one more argument than there are *list*s and returns a single value. *Combine* must not mutate the *list* arguments. The `fold-right` procedure iterates the *combine* procedure over the values of the *list*s from right to left and an accumulator value, starting with an accumulator value of *nil*. More specifically, `fold-right` returns *nil* if the *list*s are empty. If they are not empty, *combine* is first applied to the respective last elements of the *list*s in order and *nil*. The result becomes the new accumulator value, and *combine* is applied to the respective previous elements of the *list* and the new accumulator value. This step is repeated until the beginning of the list is reached; then the accumulator value is returned. *Proc* is always called in same dynamic environment as `fold-right` itself.

```
(fold-right + 0 '(1 2 3 4 5))⟹ 15

(fold-right cons '() '(1 2 3 4 5))
          ⟹ (1 2 3 4 5)

(fold-right (lambda (x l)
              (if (odd? x) (cons x l) l))
            '()
            '(3 1 4 1 5 9 2 6 5))
                        ⟹ (3 1 1 5 9 5)

(fold-right cons '(q) '(a b c))
          ⟹ (a b c q)

(fold-right + 0 '(1 2 3) '(4 5 6))
          ⟹ 21
```

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.

`(remp` *proc list*`)`	procedure
`(remove` *obj list*`)`	procedure
`(remv` *obj list*`)`	procedure
`(remq` *obj list*`)`	procedure

Proc must be a procedure that takes a single argument and returns a single value. *Proc* must not mutate the *list*

arguments. Each of these procedures returns a list of the elements of *list* that do not satisfy a given condition. The `remp` procedure successively applies *proc* to the elements of *list* and returns a list of the values of *list* for which *proc* returned `#f`. *Proc* is always called in same dynamic environment as `remp` itself. The `remove`, `remv`, and `remq` procedures return a list of the elements that are not *obj*. The `remq` procedure uses `eq?` to compare *obj* with the elements of *list*, while `remv` uses `eqv?` and `remove` uses `equal?`.

```
(remp even? '(3 1 4 1 5 9 2 6 5))
          ⟹ (3 1 1 5 9 5)

(remove 1 '(3 1 4 1 5 9 2 6 5))
          ⟹ (3 4 5 9 2 6 5)

(remv 1 '(3 1 4 1 5 9 2 6 5))
          ⟹ (3 4 5 9 2 6 5)

(remq 'foo '(bar foo baz))  ⟹ (bar baz)
```

Implementation responsibilities: The implementation must check the restrictions on *proc* to the extent performed by applying it as described.

(`memp` *proc list*)	procedure
(`member` *obj list*)	procedure
(`memv` *obj list*)	procedure
(`memq` *obj list*)	procedure

Proc must be a procedure that takes a single argument and returns a single value. *Proc* must not mutate *list*.

These procedures return the first sublist of *list* whose car satisfies a given condition, where the sublists of *lists* are the lists returned by (`list-tail` *list k*) for *k* less than the length of *list*. The `memp` procedure applies *proc* to the cars of the sublists of *list* until it finds one for which *proc* returns a true value without traversing *list* further. *Proc* is always called in same dynamic environment as `memp` itself. The `member`, `memv`, and `memq` procedures look for the first occurrence of *obj*. If *list* does not contain an element satisfying the condition, then `#f` (not the empty list) is returned. The `member` procedure uses `equal?` to compare *obj* with the elements of *list*, while `memv` uses `eqv?` and `memq` uses `eq?`.

```
(memp even? '(3 1 4 1 5 9 2 6 5))
          ⟹ (4 1 5 9 2 6 5)

(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)
```

Implementation responsibilities: The implementation must check that *list* is a chain of pairs up to the found element, or that it is indeed a list if no element is found. It should not check that it is a chain of pairs beyond the found element. The implementation must check the restrictions on *proc* to the extent performed by applying it as described.

Rationale: Although they are ordinarily used as predicates, `memp`, `member`, `memv`, `memq`, do not have question marks in their names because they return useful values rather than just `#t` or `#f`.

(`assp` *proc alist*)	procedure
(`assoc` *obj alist*)	procedure
(`assv` *obj alist*)	procedure
(`assq` *obj alist*)	procedure

Alist (for "association list") must be lists of pairs. *Proc* must be a procedure that takes a single argument and returns a single value. *Proc* must not mutate *list*.

These procedures find the first pair in *alist* whose car field satisfies a given condition, and returns that pair without traversing *alist* further. If no pair in *alist* satisfies the condition, then `#f` is returned. The `assp` procedure successively applies *proc* to the car fields of *al* and looks for a pair for which it returns a true value. *Proc* is always called in same dynamic environment as `assp` itself. The `assoc`, `assv`, and `assq` procedures look for a pair that has *obj* as its car. The `assoc` procedure uses `equal?` to compare *obj* with the car fields of the pairs in *al*, while `assv` uses `eqv?` and `assq` uses `eq?`.

Implementation responsibilities: The implementation must check that *alist* is a chain of pairs containing pairs up to the found pair, or that it is indeed a list of pairs if no element is found. It should not check that it is a chain of pairs beyond the found element. The implementation must check the restrictions on *proc* to the extent performed by applying it as described.

```
(define d '((3 a) (1 b) (4 c)))

(assp even? d)              ⟹ (4 c)
(assp odd? d)               ⟹ (3 a)

(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)
```

4. Sorting

This chapter describes the `(r6rs sorting)` library for sorting lists and vectors.

`(list-sort` *proc list*`)` procedure
`(vector-sort` *proc vector*`)` procedure

Proc must be a procedure that accepts any two elements of the list or vector. This procedure should not have any side effects. *Proc* returns a true value when its first argument is strictly less than its second, and `#f` otherwise.

The `list-sort` and `vector-sort` procedures perform a stable sort of *list* or *vector*, without changing *list* or *vector* in any way. The `list-sort` procedure returns a list, and `vector-sort` returns a vector. The results may be `eq?` to the argument when the argument is already sorted, and the result of `list-sort` may share structure with a tail of the original list. The sorting algorithm performs $O(n \lg n)$ calls to the predicate where n is the length of *list* or *vector*, and all arguments passed to the predicate are elements of the list or vector being sorted, but the pairing of arguments and the sequencing of calls to the predicate are not specified.

5. Records

This section describes abstractions for creating new data types representing records—data structures with named fields. The record mechanism comes in four libraries:

- the `(r6rs records procedural)` library, a procedural layer for creating and manipulating record types and record instances,
- the `(r6rs records explicit)` library, an explicit-naming syntactic layer for defining record types and explicitly named bindings for various procedures to manipulate the record type,
- the `(r6rs records implicit)` library, an implicit-naming syntactic layer that extends the explicit-naming syntactic layer, allowing the names of the defined procedures to be determined implicitly from the names of the record type and fields, and
- the `(r6rs records inspection)` library, a set of inspection procedures.

The procedural layer allows programs to construct new record types and the associated procedures for creating and manipulating records dynamically. It is particularly useful for writing interpreters that construct host-compatible record types. It may also serve as a target for expansion of the syntactic layers.

The explicit-naming syntactic layer provides a basic syntactic interface whereby a single record definition serves as a shorthand for the definition of several record creation and manipulation routines: a construction procedure, a predicate, field accessors, and field mutators. As the name suggests, the explicit-naming syntactic layer requires the programmer to name each of these procedures explicitly.

The implicit-naming syntactic layer extends the explicit-naming syntactic layer by allowing the names for the construction procedure, predicate, accessors, and mutators to be determined automatically from the name of the record and names of the fields. This establishes a standard naming convention and allows record-type definitions to be more succinct, with the downside that the procedure definitions cannot easily be located via a simple search for the procedure name. The programmer may override some or all of the default names by specifying them explicitly, as in the explicit-naming syntactic layer.

The two syntactic layers are designed to be fully compatible; the implicit-naming layer is simply a conservative extension of the explicit-naming layer. The design makes both explicit-naming and implicit-naming definitions natural while allowing a seamless transition between explicit and implicit naming.

Each of these layers permits record types to be extended via single inheritance, allowing record types to model hierarchies that occur in applications like algebraic data types as well as single-inheritance class systems.

Each of the layers also supports generative and nongenerative record types.

The inspection procedures allow programs to obtain from a record instance a descriptor for the type and from there obtain access to the fields of the record instance. This allows the creation of portable printers and inspectors. A program may prevent access to a record's type and thereby protect the information stored in the record from the inspection mechanism by declaring the type opaque. Thus, opacity as presented here can be used to enforce abstraction barriers.

This section uses the *rtd* and *constructor-descriptor* parameter names for arguments that must be record-type descriptors and constructor descriptors, respectively (see section 5.1).

5.1. Procedural layer

The procedural layer is provided by the `(r6rs records procedural)` library.

`(make-record-type-descriptor` *name parent uid sealed? opaque? fields*`)` procedure

Returns a *record-type descriptor*, or *rtd*, representing a record type distinct from all built-in types and other record types.

The *name* argument must be a symbol naming the record type; it is intended purely for informational purposes and may be used for printing by the underlying Scheme system.

The *parent* argument must be either `#f` or an rtd. If it is an rtd, the returned record type, t, extends the record type p represented by *parent*. Each record of type t is also a record of type p, and all operations applicable to a record of type p are also applicable to a record of type t, except for inspection operations if t is opaque but p is not. An exception with condition type `&assertion` is raised if *parent* is sealed (see below).

The extension relationship is transitive in the sense that a type extends its parent's parent, if any, and so on.

The *uid* argument must be either `#f` or a symbol. If *uid* is a symbol, the record-creation operation is *nongenerative* i.e., a new record type is created only if no previous call to `make-record-type-descriptor` was made with the *uid*. If *uid* is `#f`, the record-creation operation is *generative*, i.e., a new record type is created even if a previous call to `make-record-type-descriptor` was made with the same arguments.

If `make-record-type-descriptor` is called twice with the same *uid* symbol, the parent arguments in the two calls must be `eqv?`, the *fields* arguments `equal?`, the *sealed?* arguments boolean-equivalent (both false or both non-false), and the *opaque?* arguments boolean-equivalent. If these conditions are not met, an exception with condition type `&assertion` is raised when the second call occurs. If they are met, the second call returns, without creating a new record type, the same record-type descriptor (in the sense of `eqv?`) as the first call.

Note: Users are encouraged to use symbol names constructed using the UUID namespace (for example, using the record-type name as a prefix) for the uid argument.

The *sealed?* flag must be a boolean. If true, the returned record type is sealed, i.e., it cannot be extended.

The *opaque?* flag must be a boolean. If true, the record type is opaque. If passed an instance of the record type, `record?` returns `#f` and `record-rtd` (see "Inspection" below) raises an exception with condition type `&assertion`. The record type is also opaque if an opaque parent is supplied. If *opaque?* is false and an opaque parent is not supplied, the record is not opaque.

The *fields* argument must be a vector of field specifiers. Each field specifier must be a list of the form `(mutable` *name*`)` or a list of the form `(immutable` *name*`)`. Each name must be a symbol and names the corresponding field of the record type; the names need not be distinct. A field identified as mutable may be modified, whereas an attempt to obtain a mutator for a field identified as immutable raises an exception with condition type `&assertion`. Where field order is relevant, e.g., for record construction and field access, the fields are considered to be ordered as specified, although no particular order is required for the actual representation of a record instance.

The specified fields are added to the parent fields, if any, to determine the complete set of fields of the returned record type. If *fields* is modified after `make-record-type` has been called, the effect on the returned rtd is unspecified.

A record type is considered immutable if all fields in its complete set of fields is immutable, and is mutable otherwise.

A generative record-type descriptor created by a call to `make-record-type-descriptor` is not `eqv?` to any record-type descriptor (generative or nongenerative) created by another call to `make-record-type-descriptor`. A generative record-type descriptor is `eqv?` only to itself, i.e., `(eqv?` rtd_1 rtd_2`)` iff `(eq?` rtd_1 rtd_2`)`. Also, two nongenerative record-type descriptors are `eqv?` iff they were created by calls to `make-record-type-descriptor` with the same uid arguments.

Rationale: The record and field names passed to `make-record-type-descriptor` and appearing in the explicit-naming syntactic layer are for informational purposes only, e.g., for printers and debuggers. In particular, the accessor and mutator creation routines do not use names, but rather field indices, to identify fields.

Thus, field names are not required to be distinct in the procedural or implicit-naming syntactic layers. This relieves macros and other code generators from the need to generate distinct names.

The record and field names are used in the implicit-naming syntactic layer for the generation of accessor and mutator names, and duplicate field names may lead to accessor and mutator naming conflicts.

Rationale: Sealing a record type can help to enforce abstraction barriers by preventing extensions that may expose implementation details of the parent type. Type extensions also make monomorphic code polymorphic and difficult to change the parent class at a later time, and also prevent effective predictions of types by a compiler or human reader.

Rationale: Multiple inheritance was considered but omitted from the records facility, as it raises a number of semantic issues such as sharing among common parent types.

`(record-type-descriptor?` *obj*`)` procedure

Returns `#t` if the argument is a record-type descriptor, `#f` otherwise.

`(make-record-constructor-descriptor` *rtd* *parent-constructor-descriptor* *protocol*`)` procedure

Returns a *record-constructor descriptor* (or *constructor descriptor* for short) that can be used to create record constructors (via `record-constructor`; see below) or other

constructor descriptors. *Rtd* must be a record-type descriptor. *Protocol* must be a procedure or `#f`. If it is `#f`, a default *protocol* procedure is supplied. If *protocol* is a procedure, it is called by `record-constructor` with a single argument p and must return a procedure that creates and returns an instance of the record type using p as described below.

If *rtd* is not an extension of another record type, then *parent-constructor-descriptor* must be `#f`. In this case, *protocol*'s argument p is a procedure *new* that expects one parameter for every field of *rtd* and returns a record instance with the fields of *rtd* initialized to its arguments. The procedure returned by *protocol* may take any number of arguments but must call *new* with the number of arguments it expects and return the resulting record instance, as shown in the simple example below.

```
(lambda (new)
  (lambda (v1 ...)
    (new v1 ...)))
```

Here, the call to *new* returns a record whose fields are simply initialized with the arguments `v1 ...`. The expression above is equivalent to `(lambda (new) new)`.

If *rtd* is an extension of another record type *parent-rtd*, *parent-constructor-descriptor* must be a constructor descriptor of *parent-rtd* or `#f`. If it is `#f`, a default constructor descriptor is assumed, whose constructor accepts as many values as there are fields and initializes the fields to the arguments. In the extension case, p is a procedure that accepts the same number of arguments as the constructor of *parent-constructor-descriptor* and returns a procedure *new*, which, as above, expects one parameter for every field of *rtd* (not including parent fields) and returns a record instance with the fields of *rtd* initialized to its arguments and the fields of *parent-rtd* and its parents initialized by the constructor of *parent-constructor-descriptor*. A simple *protocol* in this case might be written as follows.

```
(lambda (p)
  (lambda (x1 ... v1 ...)
    (let ((new (p x ...)))
      (new v1 ...))))
```

This passes some number of arguments `x1 ...` to p for the constructor of *parent-constructor-descriptor* and calls *new* with `v1 ...` to initialize the child fields.

The constructor descriptors for a record type form a chain of protocols exactly parallel to the chain of record-type parents. Each constructor descriptor in the chain determines the field values for the associated record type. Child record constructors need not know the number or contents of parent fields, only the number of arguments required by the parent constructor.

protocol may be `#f`, specifying a default, only if *rtd* is not an extension of another record type, or, if it is, if the parent constructor-descriptor encapsulates a default protocol. In the first case, the default *protocol* procedure is equivalent to the following:

```
(lambda (p)
  (lambda field-values
    (apply p field-values)))
```

or, simply, `(lambda (`p`) `p`)`.

In the second case, the default *protocol* procedure returns a constructor that accepts one argument for each of the record type's complete set of fields (including those of the parent record type, the parent's parent record type, etc.) and returns a record with the fields initialized to those arguments, with the field values for the parent coming before those of the extension in the argument list.

Even if *rtd* extends another record type, *parent-constructor-descriptor* may also be `#f`, in which case a constructor with default protocol is supplied.

Rationale: The constructor-descriptor mechanism is an infrastructure for creating specialized constructors, rather than just creating default constructors that accept the initial values of all the fields as arguments. This infrastructure achieves full generality while leaving each level of an inheritance hierarchy in control over its own fields and allowing child record definitions to be abstracted away from the actual number and contents of parent fields.

The design allows the initial values of the fields to be specially computed or to default to constant values. It also allows for operations to be performed on or with the resulting record, such as the registration of a record for finalization. Moreover, the constructor-descriptor mechanism allows the creation of such initializers in a modular manner, separating the initialization concerns of the parent types from those of the extensions.

The mechanism described here achieves complete generality without cluttering the syntactic layer, sacrificing a bit of notational convenience in special cases.

`(record-constructor` *constructor-descriptor*`)` procedure

Calls the *protocol* of *constructor-descriptor* (as described for `make-record-constructor-descriptor`) and returns the resulting construction procedure *constructor* for instances of the record type associated with *constructor-descriptor*.

Two values created by *constructor* are equal according to `equal?` iff they are `eqv?`, provided their record type is not used to implement any of the types explicitly mentioned in the definition of `equal?`.

For any *constructor* returned by `record-constructor`, the following holds:

```
(let ((r (constructor v ...)))
  (eqv? r r))                ⟹ #t
```

For mutable records, but not necessarily for immutable ones, the following hold. (A record of a mutable record type is mutable; a record of an immutable record type is immutable.)

```
(let ((r (constructor v ...)))
  (eq? r r))                 ⟹ #t

(let ((f (lambda () (constructor v ...))))
  (eq? (f) (f)))             ⟹ #f
```

`(record-predicate` *rtd*`)` procedure

Returns a procedure that, given an object *obj*, returns a boolean that is `#t` iff *obj* is a record of the type represented by *rtd*.

`(record-accessor` *rtd k*`)` procedure

K must be a valid field index of *rtd*. The `record-accessor` procedure returns a one-argument procedure that, given a record of the type represented by *rtd*, returns the value of the selected field of that record.

The field selected is the one corresponding the *k*th element (0-based) of the *fields* argument to the invocation of `make-record-type-descriptor` that created *rtd*. Note that *k* cannot be used to specify a field of any type *rtd* extends.

If the accessor procedure is given something other than a record of the type represented by *rtd*, an exception with condition type `&assertion` is raised. Records of the type represented by *rtd* include records of extensions of the type represented by *rtd*.

`(record-mutator` *rtd k*`)` procedure

K must be a valid field index of *rtd*. The `record-mutator` procedure returns a two-argument procedure that, given a record *r* of the type represented by *rtd* and an object *obj*, stores *obj* within the field of *r* specified by *k*. The *k* argument is as in `record-accessor`. If *k* specifies an immutable field, an exception with condition type `&assertion` is raised. The mutator returns the unspecified value.

```
(define :point
  (make-record-type-descriptor
    'point #f
    #f #f #f
    '#((mutable x) (mutable y))))

(define make-point
  (record-constructor
    (make-record-constructor-descriptor :point
      #f #f)))

(define point? (record-predicate :point))
(define point-x (record-accessor :point 0))
(define point-y (record-accessor :point 1))
(define point-x-set! (record-mutator :point 0))
(define point-y-set! (record-mutator :point 1))

(define p1 (make-point 1 2))
(point? p1)                  ⟹ #t
(point-x p1)                 ⟹ 1
(point-y p1)                 ⟹ 2
(point-x-set! p1 5)          ⟹ the unspecified value
(point-x p1)                 ⟹ 5

(define :point2
  (make-record-type-descriptor
    'point2 :point
    #f #f #f '#((mutable x) (mutable y))))

(define make-point2
  (record-constructor
    (make-record-constructor-descriptor :point2
      #f #f)))
(define point2? (record-predicate :point2))
(define point2-xx (record-accessor :point2 0))
(define point2-yy (record-accessor :point2 1))

(define p2 (make-point2 1 2 3 4))
(point? p2)                  ⟹ #t
(point-x p2)                 ⟹ 1
(point-y p2)                 ⟹ 2
(point2-xx p2)               ⟹ 3
(point2-yy p2)               ⟹ 4
```

5.2. Explicit-naming syntactic layer

The explicit-naming syntactic layer is provided by the `(r6rs records explicit)` library.

The record-type-defining form `define-record-type` is a definition and can appear anywhere any other ⟨definition⟩ can appear.

`(define-record-type` ⟨name spec⟩ ⟨record clause⟩*`)` syntax

A `define-record-type` form defines a record type along with associated constructor descriptor and constructor, predicate, field accessors, and field mutators. The `define-record-type` form expands into a set of definitions in the environment where `define-record-type` appears; hence, it is possible to refer to the bindings (except for that of the record type itself) recursively.

The ⟨name spec⟩ specifies the names of the record type, construction procedure, and predicate. It must take the following form.

```
(⟨record name⟩ ⟨constructor name⟩ ⟨predicate name⟩)
```

⟨Record name⟩, ⟨constructor name⟩, and ⟨predicate name⟩ must all be identifiers.

⟨Record name⟩, taken as a symbol, becomes the name of the record type. Additionally, it is bound by this definition to an expand-time or run-time description of the record type for use as parent name in syntactic record-type definitions that extend this definition. It may also be used as a handle to gain access to the underlying record-type descriptor and constructor descriptor (see `record-type-descriptor` and `record-constructor-descriptor` below).

⟨Constructor name⟩ is defined by this definition to be a constructor for the defined record type, with a protocol specified by the `protocol` clause, or, in its absence, using a default protocol. For details, see the description of the `protocol` clause below.

⟨Predicate name⟩ is defined by this definition to a predicate for the defined record type.

Each ⟨record clause⟩ must take one of the following forms; it is a syntax violation if multiple ⟨record clause⟩s of the same kind appear in a `define-record-type` form.

- `(fields` ⟨field-spec⟩*`)`

  where each ⟨field-spec⟩ has one of the following forms

  ```
  (immutable ⟨field name⟩ ⟨accessor name⟩)
  (mutable ⟨field name⟩
           ⟨accessor name⟩ ⟨mutator name⟩)
  ```

  ⟨Field name⟩, ⟨accessor name⟩, and ⟨mutator name⟩ must all be identifiers. The first form declares an immutable field called ⟨field name⟩, with the corresponding accessor named ⟨accessor name⟩. The second form declares a mutable field called ⟨field name⟩, with the corresponding accessor named ⟨accessor name⟩, and with the corresponding mutator named ⟨mutator name⟩.

  The ⟨field name⟩s become, as symbols, the names of the fields of the record type being created, in the same order. They are not used in any other way.

- `(parent` ⟨parent name⟩`)`

  This specifies that the record type is to have parent type ⟨parent name⟩, where ⟨parent name⟩ is the ⟨record name⟩ of a record type previously defined using `define-record-type`. The absence of a `parent` clause implies a record type with no parent type.

- `(protocol` ⟨expression⟩`)`

  ⟨Expression⟩ is evaluated in the same environment as the define-record-type form, and must evaluate to a protocol appropriate for the record type being defined (see the description of `make-record-constructor-descriptor`). The protocol is used to create a record-constructor descriptor where, if the record type being defined has a parent, the parent-type constructor descriptor is the one associated with the parent type specified in the `parent` clause.

  If no `protocol` clause is specified, a constructor descriptor is still created using a default protocol. The rules for this are the same as for `make-record-constructor-descriptor`: the clause can be absent only if the record type defined has no parent type, or if the parent definition does not specify a protocol.

- `(sealed #t)`
  `(sealed #f)`

  If this option is specified with operand `#t`, the defined record type is sealed. Otherwise, the defined record type is not sealed.

- `(opaque #t)`
  `(opaque #f)`

  If this option is specified with operand `#t`, or if an opaque parent record type is specified, the defined record type is opaque. Otherwise, the defined record type is not opaque.

- `(nongenerative` ⟨uid⟩`)`
  `(nongenerative)`

  This specifies that the record type is nongenerative with uid ⟨uid⟩, which must be an ⟨identifier⟩. If ⟨uid⟩ is absent, a unique uid is generated at macro-expansion time. If two record-type definitions specify the same *uid*, then the implied arguments to `make-record-type-descriptor` must be equivalent as described under `make-record-type-descriptor`. If this condition is not met, it is either considered a syntax violation or an exception with condition type `&assertion` is raised. If the condition is met, a single record type is generated for both definitions.

  In the absence of a `nongenerative` clause, a new record type is generated every time a `define-record-type` form is evaluated:

  ```
  (let ((f (lambda (x)
             (define-record-type r ...)
             (if x r? (make-r ...)))))
    ((f #t) (f #f)))          ⟹ #f
  ```

All bindings created by `define-record-type` (for the record type, the construction procedure, the predicate, the accessors, and the mutators) must have names that are pairwise distinct.

`(record-type-descriptor` ⟨record name⟩`)` syntax

Evaluates to the record-type descriptor associated with the type specified by ⟨record-name⟩.

Note that `record-type-descriptor` works on both opaque and non-opaque record types.

`(record-constructor-descriptor` ⟨record name⟩`)` syntax

Evaluates to the record-constructor descriptor associated with ⟨record name⟩.

Explicit-naming syntactic-layer examples:

```
(define-record-type (point3 make-point3 point3?)
  (fields (immutable x point3-x)
          (mutable y point3-y set-point3-y!))
  (nongenerative
    point3-4893d957-e00b-11d9-817f-00111175eb9e))

(define-record-type (cpoint make-cpoint cpoint?)
  (parent point3)
  (protocol
   (lambda (p)
     (lambda (x y c)
       ((p x y) (color->rgb c)))))
  (fields
    (mutable rgb cpoint-rgb cpoint-rgb-set!)))

(define (color->rgb c)
  (cons 'rgb c))

(define p3-1 (make-point3 1 2))
(define p3-2 (make-cpoint 3 4 'red))

(point3? p3-1)              ⟹ #t
(point3? p3-2)              ⟹ #t
(point3? (vector))          ⟹ #f
(point3? (cons 'a 'b))      ⟹ #f
(cpoint? p3-1)              ⟹ #f
(cpoint? p3-2)              ⟹ #t
(point3-x p3-1)             ⟹ 1
(point3-y p3-1)             ⟹ 2
(point3-x p3-2)             ⟹ 3
(point3-y p3-2)             ⟹ 4
(cpoint-rgb p3-2)           ⟹ '(rgb . red)

(set-point3-y! p3-1 17)
(point3-y p3-1)             ⟹ 17)

(record-rtd p3-1)
         ⟹ (record-type-descriptor point3)

(define-record-type (ex1 make-ex1 ex1?)
  (protocol (lambda (new) (lambda a (new a))))
  (fields (immutable f ex1-f)))

(define ex1-i1 (make-ex1 1 2 3))
(ex1-f ex1-i1)              ⟹ '(1 2 3)

(define-record-type (ex2 make-ex2 ex2?)
  (protocol
    (lambda (new) (lambda (a . b) (new a b))))
  (fields (immutable a ex2-a)
          (immutable b ex2-b)))

(define ex2-i1 (make-ex2 1 2 3))
(ex2-a ex2-i1)              ⟹ 1
(ex2-b ex2-i1)              ⟹ '(2 3)

(define-record-type (unit-vector
                     make-unit-vector
                     unit-vector?)
  (protocol
   (lambda (new)
     (lambda (x y z)
       (let ((length
              (sqrt (+ (* x x) (* y y) (* z z)))))
         (new (/ x length)
              (/ y length)
              (/ z length))))))
  (fields (immutable x unit-vector-x)
          (immutable y unit-vector-y)
          (immutable z unit-vector-z)))
```

5.3. Implicit-naming syntactic layer

The implicit-naming syntactic layer is provided by the `(r6rs records implicit)` library.

The `define-record-type` form of the implicit-naming syntactic layer is a conservative extension of the `define-record-type` form of the explicit-naming layer: a `define-record-type` form that conforms to the syntax of the explicit-naming layer also conforms to the syntax of the implicit-naming layer, and any definition in the implicit-naming layer can be understood by its translation into the explicit-naming layer.

This means that a record type defined by the `define-record-type` form of either layer can be used by the other.

The implicit-naming syntactic layer extends the explicit-naming layer in two ways. First, ⟨name-spec⟩ may be a single identifier representing just the record name. In this case, the name of the construction procedure is generated by prefixing the record name with `make-`, and the predicate name is generated by adding a question mark (`?`) to the end of the record name. For example, if the record

name is `frob`, the name of the construction procedure is `make-frob`, and the predicate name is `frob?`.

Second, the syntax of ⟨field-spec⟩ is extended to allow the accessor and mutator names to be omitted. That is, ⟨field-spec⟩ can take one of the following forms as well as the forms described in the preceding section.

```
(immutable ⟨field name⟩)
(mutable ⟨field name⟩)
```

If ⟨field-spec⟩ takes one of these forms, the accessor name is generated by appending the record name and field name with a hyphen separator, and the mutator name (for a mutable field) is generated by adding a `-set!` suffix to the accessor name. For example, if the record name is `frob` and the field name is `widget`, the accessor name is `frob-widget` and the mutator name is `frob-widget-set!`.

Third, the syntax of ⟨field-spec⟩ is also extended to consist of just the field name. The corresponding field is assumed to be immutable.

Any definition that takes advantage of implicit naming can be rewritten trivially to a definition that conforms to the syntax of the explicit-naming layer merely by specifying the names explicitly. For example, the implicit-naming layer record definition:

```
(define-record-type frob
  (fields (mutable widget))
  (protocol
    (lambda (c) (c (make-widget n)))))
```

is equivalent to the following explicit-naming layer record definition.

```
(define-record-type (frob make-frob frob?)
  (fields (mutable widget
                   frob-widget frob-widget-set!))
  (protocol
    (lambda (c) (c (make-widget n)))))
```

Also, the implicit-naming layer record definition:

```
(define-record-type point (fields x y))
```

is equivalent to the following explicit-naming layer record definition:

```
(define-record-type (point make-point point?)
  (fields
    (immutable x point-x)
    (immutable y point-y)))
```

With the implicit-naming layer, one can choose to specify just some of the names explicitly; for example, the following overrides the choice of accessor and mutator names for the widget field.

```
(define-record-type frob
  (fields (mutable widget getwid setwid!))
  (protocol
    (lambda (c) (c (make-widget n)))))


(define *ex3-instance* #f)

(define-record-type ex3
  (parent cpoint)
  (protocol
   (lambda (p)
     (lambda (x y t)
       (let ((r ((p x y 'red) t)))
         (set! *ex3-instance* r)
         r))))
  (fields
   (mutable thickness))
  (sealed #t) (opaque #t))

(define ex3-i1 (make-ex3 1 2 17))
(ex3? ex3-i1)                ⟹ #t
(cpoint-rgb ex3-i1)          ⟹ '(rgb . red)
(ex3-thickness ex3-i1)       ⟹ 17
(ex3-thickness-set! ex3-i1 18)
(ex3-thickness ex3-i1)       ⟹ 18
*ex3-instance*               ⟹ ex3-i1

(record? ex3-i1)             ⟹ #f
```

`(record-type-descriptor` ⟨record name⟩`)` syntax

This is the same as `record-type-descriptor` from the `(r6rs records explicit)` library.

`(record-constructor-descriptor` ⟨record name⟩`)` syntax

This is the same as `record-constructor-descriptor` from the `(r6rs records explicit)` library.

5.4. Inspection

The implicit-naming syntactic layer is provided by the `(r6rs records inspection)` library.

A set of procedures are provided for inspecting records and their record-type descriptors. These procedures are designed to allow the writing of portable printers and inspectors.

Note that `record?` and `record-rtd` treat records of opaque record types as if they were not records. On the other hand, the inspection procedures that operate on record-type descriptors themselves are not affected by opacity. In other words, opacity controls whether a program can obtain an rtd from an instance. If the program has access

to the original rtd via `make-record-type-descriptor` or `record-type-descriptor`, it can still make use of the inspection procedures.

Any of the standard types mentioned in this report may or may not be implemented as an opaque record type. Consequently, `record?`, when applied to an object of one of these types, may return `#t`. In this case, inspection is possible for these objects.

`(record?` *obj*`)` procedure

Returns `#t` if *obj* is a record, and its record type is not opaque. Returns `#f` otherwise.

`(record-rtd` *record*`)` procedure

Returns the rtd representing the type of *record* if the type is not opaque. The rtd of the most precise type is returned; that is, the type t such that *record* is of type t but not of any type that extends t. If the type is opaque, an exception is raised with condition type `&assertion`.

`(record-type-name` *rtd*`)` procedure

Returns the name of the record-type descriptor *rtd*.

`(record-type-parent` *rtd*`)` procedure

Returns the parent of the record-type descriptor *rtd*, or `#f` if it has none.

`(record-type-uid` *rtd*`)` procedure

Returns the uid of the record-type descriptor rtd, or `#f` if it has none. (An implementation may assign a generated uid to a record type even if the type is generative, so the return of a uid does not necessarily imply that the type is nongenerative.)

`(record-type-generative?` *rtd*`)` procedure

Returns `#t` if *rtd* is generative, and `#f` if not.

`(record-type-sealed?` *rtd*`)` procedure

Returns a boolean value indicating whether the record-type descriptor is sealed.

`(record-type-opaque?` *rtd*`)` procedure

Returns a boolean value indicating whether the record-type descriptor is opaque.

`(record-type-field-names` *rtd*`)` procedure

Returns a vector of symbols naming the fields of the type represented by *rtd* (not including the fields of parent types) where the fields are ordered as described under `make-record-type-descriptor`. The returned vector may be immutable. If the returned vector is modified, the effect on *rtd* is unspecified.

`(record-field-mutable?` *rtd k*`)` procedure

Returns a boolean value indicating whether the field specified by k of the type represented by *rtd* is mutable, where k is as in `record-accessor`.

6. Exceptions and conditions

Scheme allows programs to deal with exceptional situations using two cooperating facilities: The exception system for raising and handling exceptional situations, and the condition system for describing these situations.

The exception system allows the program, when it detects an exceptional situation, to pass control to an exception handler, and for dynamically establishing such exception handlers. Exception handlers are always invoked with an object describing the exceptional situation. Scheme's condition system provides a standardized taxonomy of such descriptive objects, as well as a facility for extending the taxonomy.

6.1. Exceptions

This section describes Scheme's exception-handling and exception-raising constructs provided by the `(r6rs exceptions)` library.

Note: This specification follows SRFI 34 [6].

Exception handlers are one-argument procedures that determine the action the program takes when an exceptional situation is signalled. The system implicitly maintains a current exception handler.

The program raises an exception by invoking the current exception handler, passing to it an object encapsulating information about the exception. Any procedure accepting one argument may serve as an exception handler and any object may be used to represent an exception.

The system maintains the current exception handler as part of the *dynamic environment* of the program, the context for `dynamic-wind`. The dynamic environment can be thought of as that part of a continuation that does not specify the destination of any returned values. It includes the `dynamic-wind` context and the current exception handler.

When a program begins its execution, the current exception handler is expected to handle all `&serious` conditions by interrupting execution, reporting that an exception has been raised, and displaying information about the condition object that was provided. The handler may then exit, or may provide a choice of other options. Moreover, the

exception handler is expected to return when passed any other non-`&serious` condition. Interpretation of these expectations necessarily depends upon the nature of the system in which programs are executed, but the intent is that users perceive the raising of an exception as a controlled escape from the situation that raised the exception, not as a crash.

`(with-exception-handler` *handler thunk*`)` procedure

Handler must be a procedure that accepts one argument. The `with-exception-handler` procedure returns the result(s) of invoking *thunk*. *Handler* is installed as the current exception handler for the dynamic extent (as determined by `dynamic-wind`) of the invocation of *thunk*.

`(guard (`⟨variable⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` ⟨body⟩`)` syntax

Syntax: Each ⟨clause⟩ must have the same form as a `cond` clause. (See report section 9.5.5.)

Semantics: Evaluating a `guard` form evaluates ⟨body⟩ with an exception handler that binds the raised object to ⟨variable⟩ and within the scope of that binding evaluates the clauses as if they were the clauses of a `cond` expression. That implicit `cond` expression is evaluated with the continuation and dynamic environment of the `guard` expression. If every ⟨clause⟩'s ⟨test⟩ evaluates to false and there is no else clause, then `raise` is re-invoked on the raised object within the dynamic environment of the original call to raise except that the current exception handler is that of the `guard` expression.

`(raise` *obj*`)` procedure

Raises a non-continuable exception by invoking the current exception handler on *obj*. The handler is called with a continuation whose dynamic environment is that of the call to `raise`, except that the current exception handler is the one that was in place when the handler being called was installed. The continuation of the handler raises a non-continuable exception with condition type `&non-continuable`.

`(raise-continuable` *obj*`)` procedure

Raises a continuable exception by invoking the current exception handler on *obj*. The handler is called with a continuation that is equivalent to the continuation of the call to `raise-continuable` with these two exceptions: (1) the current exception handler is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to `raise-continuable`.

```
(guard (con
        ((error? con)
         (if (message-condition? con)
             (display (condition-message con))
             (display "an error has occurred")
             'error))
        ((violation? con)
         (if (message-condition? con)
             (display (condition-message con))
             (display "the program has a bug"))
         'violation))
  (raise
    (condition
      (&error)
      (&message (message "I am an error")))))
    prints: I am an error
                              ⟹ error
(guard (con
        ((error? con)
         (if (message-condition? con)
             (display (condition-message con))
             (display "an error has occurred")
             'error)))
  (raise
    (condition
      (&violation)
      (&message (message "I am an error")))))
                              ⟹ &violation exception

(guard (con
        ((error? con)
         (display "error opening file")
         #f))
  (call-with-input-file "foo.scm" read))
    prints: error opening file
                              ⟹ #f

(with-exception-handler
  (lambda (con)
    (cond
      ((not (warning? con))
       (raise con))
      ((message-condition? con)
       (display (condition-message con)))
      (else
       (display "a warning has been issued")))
    42)
  (lambda ()
    (+ (raise-continuable
         (condition
           (&warning)
           (&message
             (message "should be a number"))))
       23)))
    prints: should be a number
                              ⟹ 65
```

6.2. Conditions

The section describes Scheme `(r6rs conditions)` library for creating and inspecting condition types and values. A condition value encapsulates information about an exceptional situation, or *exception*. Scheme also defines a number of basic condition types.

Note: This specification is similar to, but not identical with SRFI 35 [7].

Scheme conditions provides two mechanisms to enable communication about exceptional situation: subtyping among condition types allows handling code to determine the general nature of an exception even though it does not anticipate its exact nature, and compound conditions allow an exceptional situation to be described in multiple ways.

Rationale: Conditions are values that communicate information about exceptional situations between parts of a program. Code that detects an exception may be in a different part of the program than the code that handles it. In fact, the former may have been written independently from the latter. Consequently, to facilitate effective handling of exceptions, conditions must communicate as much information as possible as accurately as possible, and still allow effective handling by code that did not precisely anticipate the nature of the exception that occurred.

6.2.1. Condition objects

Conditions are objects with named fields. Each condition belongs to one or more condition types. Each condition type specifies a set of field names. A condition belonging to a condition type includes a value for each of the type's field names. These values can be extracted from the condition by using the appropriate field name.

The condition system distinguishes between *simple conditions* and *compound conditions*. A compound condition consists of an ordered set of simple conditions. Thus, every condition can be viewed as an ordered set of simple component conditions: If it is simple, the set consists of the condition itself; if it is compound, it consists of the simple conditions that compose it.

There is a tree of condition types with the distinguished `&condition` as its root. All other condition types have a parent condition type.

`(make-condition-type` *id parent field-names*`)` procedure

Returns a new condition type. *Id* must be a symbol that serves as a symbolic name for the condition type. *Parent* must itself be a condition type. *Field-names* must be a list of symbols. It identifies the fields of the conditions associated with the condition type.

Field-names must be disjoint from the field names of *parent* and its ancestors.

`(condition-type?` *thing*`)` procedure

Returns `#t` if *thing* is a condition type, and `#f` otherwise.

`(make-condition` *type alist*`)` procedure

Returns a simple condition value belonging to condition type *type*. *Alist* must be an association list mapping field names to arbitrary values. There must be a pair in the association list for each field of *type* and its direct and indirect supertypes. The `make-condition` procedure returns the condition value, which fields and values as indicated by *alist*.

`(condition?` *obj*`)` procedure

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

`(condition-has-type?` *condition condition-type*`)` procedure

The `condition-has-type?` procedure tests if condition condition belongs to condition type *condition-type*. It returns `#t` if any of condition's types includes *condition-type*, either directly or as an ancestor, and `#f` otherwise.

`(condition->list` *condition*`)` procedure

Returns a list of the component conditions of *condition*.

`(condition-ref` *condition type field-name*`)` procedure

Condition must be a condition, *type* a condition type, and *field-name* a symbol naming a field of type or its direct or indirect supertypes. Moreover, condition must be a simple condition of type *type*, or a compound condition containing a simple condition of type *type*. The `condition-ref` procedure returns the value of the field named by *field-name* in the first component condition of *condition* that has type *type*.

`(make-compound-condition` $condition_1$ ...`)` procedure

Returns a compound condition consisting of the component conditions of $condition_1$, ..., in that order.

`(define-condition-type` ⟨condition-type⟩ syntax
⟨supertype⟩
⟨predicate⟩
⟨field-spec$_1$⟩ ...`)`

Syntax: ⟨Condition-type⟩, ⟨supertypes⟩, and ⟨predicate⟩ must all be identifiers. Each ⟨field-spec⟩ must be of the form

(⟨field⟩ ⟨accessor⟩)

where both ⟨field⟩ and ⟨accessor⟩ must be identifiers.

Semantics: The `define-condition-type` form expands into a set of definitions:

- ⟨Condition-type⟩, which is bound to some value describing a new condition type. ⟨Supertype⟩ must be the name of a previously defined condition type.
- ⟨Predicate⟩ is bound to a predicate that identifies conditions associated with that type, or with any of its subtypes.
- Each ⟨accessor⟩ is bound to a procedure which extracts the value of the named field from a condition associated with this condition type.

`(condition` ⟨type-field-binding$_1$⟩ ...`)` syntax

Returns a condition value. Each ⟨type-field-binding⟩ must be of the form

(⟨condition-type⟩ ⟨field-binding$_1$⟩ ...)

Each ⟨field-binding⟩ must be of the form

(⟨field⟩ ⟨expression⟩)

where ⟨field⟩ is a field identifier from the definition of ⟨condition-type⟩.

The condition returned by condition is created by a call of the form

```
(make-compound-condition
  (make-condition ⟨condition-type⟩
                  (list (cons '⟨field-name⟩ ⟨expression⟩)
                        ...))
  ...)
```

with the condition types retaining their order from the condition form.

`&condition` condition type

This is the root of the entire condition type hierarchy. It has no fields.

```
(define-condition-type &c &condition
  c?
  (x c-x))

(define-condition-type &c1 &c
  c1?
  (a c1-a))

(define-condition-type &c2 &c
  c2?
  (b c2-b))
```

```
(define v1
  (make-condition &c1
    (list (cons 'x "V1")
          (cons 'a "a1"))))

(c? v1)                    ⟹ #t
(c1? v1)                   ⟹ #t
(c2? v1)                   ⟹ #f
(c-x v1)                   ⟹ "V1"
(c1-a v1)                  ⟹ "a1"

(define v2 (condition (&c2
                        (x "V2")
                        (b "b2"))))

(c? v2)                    ⟹ #t
(c1? v2)                   ⟹ #f
(c2? v2)                   ⟹ #t
(c-x v2)                   ⟹ "V2"
(c2-b v2)                  ⟹ "b2"

(define v3 (condition (&c1
                       (x "V3/1")
                       (a "a3"))
                      (&c2
                       (x "V3/2")
                       (b "b3"))))

(c? v3)                    ⟹ #t
(c1? v3)                   ⟹ #t
(c2? v3)                   ⟹ #t
(c-x v3)                   ⟹ "V3/1"
(c1-a v3)                  ⟹ "a3"
(c2-b v3)                  ⟹ "b3"

(define v4 (make-compound-condition v1 v2))

(c? v4)                    ⟹ #t
(c1? v4)                   ⟹ #t
(c2? v4)                   ⟹ #t
(c-x v4)                   ⟹ "V1"
(c1-a v4)                  ⟹ "a1"
(c2-b v4)                  ⟹ "b2"

(define v5 (make-compound-condition v2 v3))

(c? v5)                    ⟹ #t
(c1? v5)                   ⟹ #t
(c2? v5)                   ⟹ #t
(c-x v5)                   ⟹ "V2"
(c1-a v5)                  ⟹ "a3"
(c2-b v5)                  ⟹ "b2"
```

6.3. Standard condition types

`&message` condition type
`(message-condition?` *obj*`)` procedure
`(condition-message` *condition*`)` procedure

This condition type could be defined by

```
(define-condition-type &message &condition
  message-condition?
  (message condition-message))
```

It carries a message further describing the nature of the condition to humans.

`&warning` condition type
`(warning?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &warning &condition
  warning?)
```

This type describes conditions that do not, in principle, prohibit immediate continued execution of the program, but may interfere with the program's execution later.

`&serious` condition type
`(serious-condition?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &serious &condition
  serious-condition?)
```

This type describes conditions serious enough that they cannot safely be ignored. This condition type is primarily intended as a supertype of other condition types.

`&error` condition type
`(error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &error &serious
  error?)
```

This type describes errors, typically caused by something that has gone wrong in the interaction of the program with the external world or the user.

`&violation` condition type
`(violation?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &violation &serious
  violation?)
```

This type describes violations of the language standard or a library standard, typically caused by a programming error.

`&non-continuable` condition type
`(non-continuable?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &non-continuable &violation
  non-continuable?)
```

This type denotes that an exception handler invoked via `raise` has returned.

`&implementation-restriction` condition type
`(implementation-restriction?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &implementation-restriction
    &violation
  implementation-restriction?)
```

This type describes a violation of an implementation restriction allowed by the specification, such as the absence of representations for NaNs and infinities. (See section 9.2.)

`&lexical` condition type
`(lexical-violation?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &lexical &violation
  lexical-violation?)
```

This type describes syntax violations at the level of the read syntax.

`&syntax` condition type
`(syntax-violation?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &syntax &violation
  syntax-violation?
  (form syntax-violation-form)
  (subform syntax-violation-subform))
```

This type describes syntax violations. The `form` field contains the erroneous syntax object or a datum representing the code of the erroneous form. The `subform` field may contain an optional syntax object or datum within the erroneous form that more precisely locates the violation. It can be `#f` to indicate the absence of more precise information.

`&undefined` condition type
`(undefined-violation?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &undefined &violation
  undefined-violation?)
```

This type describes unbound identifiers in the program.

`&assertion` condition type
`(assertion-violation?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &assertion &violation
  assertion-violation?)
```

This type describes an invalid call to a procedure, either passing an invalid number of arguments, or passing an argument of the wrong type.

`&irritants` condition type
`(irritants-condition?` *obj*`)` procedure
`(condition-irritants` *condition*`)` procedure

This condition type could be defined by

```
(define-condition-type &irritants &condition
  irritants-condition?
  (irritants condition-irritants))
```

The `irritants` field should contain a list of objects. This condition provides additional information about a condition, typically the argument list of a procedure that detected an exception. Conditions of this type are created by the `error` and `assertion-violation` procedures of report section 9.16.

`&who` condition type
`(who-condition?` *obj*`)` procedure
`(condition-who` *condition*`)` procedure

This condition type could be defined by

```
(define-condition-type &who &condition
  who-condition?
  (who condition-who))
```

The `who` field should contain a symbol or string identifying the entity reporting the exception. Conditions of this type are created by the `error` and `assertion-violation` procedures (report section 9.16), and the `syntax-violation` procedure (section 10.9).

7. I/O

This chapter describes Scheme's libraries for performing input and output:

- The `(r6rs i/o ports)` library (section 7.2) is an I/O layer for conventional, imperative buffered input and output with mixed text and binary data.
- The `(r6rs i/o simple)` library (section 7.3) is a convenience library atop the `(r6rs i/o ports)` library for textual I/O, compatible with the traditional Scheme I/O procedures [5].

Section 7.1 defines a condition-type hierarchy that is exported by both the `(r6rs i/o ports)` and `(r6rs i/o simple)` libraries.

7.1. Condition types

In exceptional situations arising from "I/O errors", the procedures described in this chapter raise an exception with condition type `&i/o`. Except where explicitly specified, there is no guarantee that the raised condition object contains all the information that would be applicable. It is recommended, however, that an implementation collect all information that is available about an exceptional situation at the place where it is detected and place it in the condition object.

The condition types and corresponding predicates and accessors are exported by both the `(r6rs i/o ports)` and `(r6rs i/o simple)` libraries. They are also exported by the `(r6rs files)` library described in chapter 8.

`&i/o` condition type
`(i/o-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o &error
  i/o-error?)
```

This is a supertype for a set of more specific I/O errors.

`&i/o-read` condition type
`(i/o-read-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-read &i/o
  i/o-read-error?)
```

This condition type describes read errors that occurred during an I/O operation.

`&i/o-write` condition type
`(i/o-write-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-write &i/o
  i/o-write-error?)
```

This condition type describes write errors that occurred during an I/O operation.

`&i/o-invalid-position` condition type
`(i/o-invalid-position-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-invalid-position &i/o
  i/o-invalid-position-error?
  (position i/o-error-position))
```

This condition type describes attempts to set the file position to an invalid position. The value of the position field is the file position that the program intended to set. This condition describes a range error, but not an assertion violation.

`&i/o-filename` condition type
`(i/o-filename-error?` *obj*`)` procedure
`(i/o-error-filename` *condition*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-filename &i/o
  i/o-filename-error?
  (filename i/o-error-filename))
```

This condition type describes an I/O error that occurred during an operation on a named file. Condition objects belonging to this type must specify a file name in the `filename` field.

`&i/o-file-protection` condition type
`(i/o-file-protection-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-file-protection
    &i/o-filename
  i/o-file-protection-error?)
```

A condition of this type specifies that an operation tried to operate on a named file with insufficient access rights.

`&i/o-file-is-read-only` condition type
`(i/o-file-is-read-only-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-file-is-read-only
    &i/o-file-protection
  i/o-file-is-read-only-error?)
```

A condition of this type specifies that an operation tried to operate on a named read-only file under the assumption that it is writeable.

`&i/o-file-already-exists` condition type
`(i/o-file-already-exists-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-file-already-exists
    &i/o-filename
  i/o-file-already-exists-error?)
```

A condition of this type specifies that an operation tried to operate on an existing named file under the assumption that it did not exist.

`&i/o-file-exists-not` condition type
`(i/o-exists-not-error?` *obj*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-file-exists-not
    &i/o-filename
  i/o-file-exists-not-error?)
```

A condition of this type specifies that an operation tried to operate on an non-existent named file under the assumption that it existed.

`&i/o-port` condition type
`(i/o-port-error?` *obj*`)` procedure
`(i/o-error-port` *condition*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-port &i/o
  i/o-port-error?
  (port i/o-error-port))
```

This condition type specifies the port with which an I/O error is associated. Except for condition objects provided for encoding and decoding errors, conditions raised by procedures may include an `&i/o-port-error` condition, but are not required to do so.

7.2. Port I/O

The `(r6rs i/o ports)` library defines an I/O layer for conventional, imperative buffered input and output. A *port* represents a buffered access object for a data sink or source or both simultaneously. The library allows ports to be created from arbitrary data sources and sinks.

The `(r6rs i/o ports)` library distinguishes between *input ports* and *output ports*. An input port is a source for data, whereas an output port is a sink for data. A port may be both an input port and an output port; such a port typically provides simultaneous read and write access to a file or other data.

The `(r6rs i/o ports)` library also distinguishes between *binary ports*, which are sources or sinks for uninterpreted bytes, and *textual ports*, which are sources or sinks for characters and strings.

This section uses *input-port*, *output-port*, *binary-port*, *textual-port*, *binary-input-port*, *textual-input-port*, *binary-output-port*, *textual-output-port*, and *port* as parameter names for arguments that must be input ports (or combined input/output ports), output ports

(or combined input/output ports), binary ports, textual ports, binary input ports, textual input ports, binary output ports, textual output ports, or any kind of port, respectively.

7.2.1. File names

Some of the procedures described in this chapter accept a file name as an argument. Valid values for such a file name include strings that name a file using the native notation of filesystem paths on an implementation's underlying operating system, and may include implementation-dependent values as well.

Rationale: Implementation-dependent file names may provide a more abstract and/or more general representation. Indeed, most operating systems do not use strings for representing file names, but rather byte or word sequences. Furthermore the string notation is not fully portable across operating systems, and is difficult to manipulate.

A *filename* parameter name means that the corresponding argument must be a file name.

7.2.2. File options

When opening a file, the various procedures in this library accept a `file-options` object that encapsulates flags to specify how the file is to be opened. A `file-options` object is an enum-set (see chapter 12) over the symbols constituting valid file options. A *file-options* parameter name means that the corresponding argument must be a file-options object.

`(file-options` ⟨file-options name⟩ ...`)` syntax

Each ⟨file-options name⟩ must be an ⟨identifier⟩. The `file-options` syntax returns a file-options object that encapsulates the specified options. The following options, which affect output only, have standard meanings:

- `create` create file if it does not already exist
- `exclusive` an exception with condition type `&i/o-file-already-exists` is raised if this option and `create` are both set and the file already exists
- `truncate` file is truncated to zero length

⟨Identifiers⟩s other than those listed above may be used as ⟨file-options name⟩s; they have implementation-specific meaning, if any.

When supplied to an operation that opens a file for output, the file-options object returned by `(file-options)` specifies that the file must already exist (otherwise an exception with condition type `&i/o-file-exists-not` is raised) and its data are unchanged by the operation; note that this default seldom coincides with the desired semantics.

Rationale: The flags specified above represent only a common subset of meaningful options on popular platforms. The `file-options` form does not restrict the ⟨file-options name⟩s, so implementations can extend the file options by platform-specific flags.

7.2.3. Buffer modes

Each output port has an associated buffer mode that defines when an output operation flushes the buffer associated with the output port. The possible buffer modes are the symbols `none` for no buffering, `line` for flushing upon line feeds and line separators (U+2028), and `block` for arbitrary buffering. This section uses the parameter name *buffer-mode* for arguments that must be buffer-mode symbols.

`(buffer-mode` ⟨name⟩`)` syntax

⟨Name⟩ must be one of the ⟨identifier⟩s `none`, `line`, or `block`. The result is the corresponding symbol, denoting the associated buffer mode.

It is a syntax violation if ⟨name⟩ is not one of the valid identifiers.

`(buffer-mode?` *obj*`)` procedure

Returns `#t` if the argument is a valid buffer-mode symbol, `#f` otherwise.

7.2.4. Transcoders

Several different Unicode encoding schemes describe standard ways to encode characters and strings as byte sequences and to decode those sequences [10]. Within this document, a *codec* is an immutable Scheme object that represents a Unicode or similar encoding scheme.

A *transcoder* is an immutable Scheme object that combines a codec with an end-of-line convention and a method for handling decoding errors. Each transcoder represents some specific bidirectional (but not necessarily lossless), possibly stateful translation between byte sequences and Unicode characters and strings. Every transcoder can operate in the input direction (bytes to characters) or in the output direction (characters to bytes), but the composition of those directions need not be identity (and often isn't). The composition of two transcoders is not defined. A *transcoder* parameter name means that the corresponding argument must be a transcoder.

Every port has a single transcoder associated with it.

A binary port is defined as a port whose associated transcoder is the *binary transcoder*, which is a special pseudo-transcoder whose input and output directions translate no byte sequences to characters, and no character sequences to bytes.

A textual port is defined as a port whose associated transcoder is not the binary transcoder.

`(latin-1-codec)` procedure
`(utf-8-codec)` procedure
`(utf-16-codec)` procedure
`(utf-32-codec)` procedure

These are predefined codecs for the ISO 8859-1, UTF-8, UTF-16, and UTF-32 encoding schemes [10].

A call to any of these procedures returns a value that is equal in the sense of `eqv?` to the result of any other call to the same procedure.

`(eol-style` *name*`)` syntax

If *name* is one of the ⟨identifier⟩s `lf`, `cr`, `crlf`, or `ls`, then the form evaluates to the corresponding symbol. If *name* is not one of these identifiers, effect and result are implementation-dependent: The result may be an eol-style symbol acceptable as an *eol-mode* argument to `make-transcoder`. Otherwise, an exception is raised.

Rationale: End-of-line styles other than those listed might become commonplace in the future.

`(native-eol-style)` procedure

Returns the default end-of-line style of the underlying platform, e.g. `lf` on Unix and `crlf` on Windows.

`&i/o-decoding` condition type
`(i/o-decoding-error?` *obj*`)` procedure

```
(define-condition-type &i/o-decoding &i/o-port
  i/o-decoding-error?
  (transcoder i/o-decoding-error-transcoder))
```

An exception with this type is raised when one of the operations for textual input from a port encounters a sequence of bytes that cannot be translated into a character or string by the input direction of the port's transcoder. The `transcoder` field contains the port's transcoder.

Exceptions of this type raised by the operations described in this section are continuable. When such an exception is raised, the port's position is at the beginning of the invalid encoding. If the exception handler returns, it must return a character or string representing the decoded text starting at the port's current position, and the exception handler must update the port's position to point past the error.

`&i/o-encoding` condition type
`(i/o-encoding-error?` *obj*`)` procedure
`(i/o-encoding-error-char` *condition*`)` procedure
`(i/o-encoding-error-transcoder` *condition*`)` procedure

This condition type could be defined by

```
(define-condition-type &i/o-encoding &i/o-port
  i/o-encoding-error?
  (char i/o-encoding-error-char)
  (transcoder i/o-encoding-error-transcoder))
```

An exception with this type is raised when one of the operations for textual output to a port encounters a character that cannot be translated into bytes by the output direction of the port's transcoder. The `char` field of the condition object contains the character that could not be encoded, and the `transcoder` field contains the transcoder associated with the port.

Exceptions of this type raised by the operations described in this section are continuable. The handler, if it returns, is expected to output to the port an appropriate encoding for the character that caused the error. The operation that raised the exception continues after that character.

`(error-handling-mode` *name*`)` syntax

If *name* is one of the ⟨identifier⟩s `ignore`, `raise`, or `replace`, then the result is the corresponding symbol. If *name* is not one of these identifiers, effect and result are implementation-dependent: The result may be an error-handling-mode symbol acceptable as a *handling-mode* argument to `make-transcoder`. Otherwise, an exception is raised.

Rationale: Implementations may support error-handling modes other than those listed.

The error-handling mode of a transcoder specifies the behavior of textual I/O operations in the presence of encoding or decoding errors.

If a textual input operation encounters an invalid or incomplete character encoding, and the error-handling mode is `ignore`, then an appropriate number of bytes of the invalid encoding are ignored and decoding continues with the following bytes. If the error-handling mode is `replace`, then the replacement character U+FFFD is injected into the data stream, an appropriate number of bytes are ignored, and decoding continues with the following bytes. If the error-handling mode is `raise`, then a continuable exception with condition type `&i/o-decoding` is raised; see the description of `&i/o-decoding` for details on how to handle such an exception.

If a textual output operation encounters a character it cannot encode, and the error-handling mode is `ignore`, then the character is ignored and encoding continues

with the next character. If the error-handling mode is `replace`, a codec-specific replacement character is emitted by the transcoder, and encoding continues with the next character. The replacement character is U+FFFD for transcoders whose codec is one of the Unicode encodings, but is the `?` character for the Latin-1 encoding. If the error-handling mode is `raise`, an exception with condition type `&i/o-encoding` is raised; see the description of `&i/o-decoding` for details on how to handle such an exception.

`(make-transcoder` *codec*`)` procedure
`(make-transcoder` *codec eol-style*`)` procedure
`(make-transcoder` *codec eol-style handling-mode*`)` procedure

Codec must be a codec; *eol-style*, if present, an eol-style symbol; and *handling-mode*, if present, an error-handling-mode symbol. *eol-style* may be omitted, in which case it defaults to the native end-of-line style of the underlying platform. *handling-mode* may be omitted, in which case it defaults to `raise`. The result is a transcoder with the behavior specified by its arguments.

A transcoder returned by `make-transcoder` is equal in the sense of `eqv?` (but not necessarily in the sense of `eq?`) to another transcoder returned by `make-transcoder` if and only if the *code*, *eol-style*, and *handling-mode* arguments are equal in the sense of `eqv?`.

`(binary-transcoder)` procedure

Returns the binary transcoder, which is equal in the sense of `eqv?` (but not necessarily in the sense of `eq?`) to the result of any call to `binary-transcoder`, and is not equal to the result of any call to `make-transcoder`.

`(transcoder-codec` *transcoder*`)` procedure
`(transcoder-eol-style` *transcoder*`)` procedure
`(transcoder-error-handling-mode` *transcoder*`)` procedure

These are accessors for transcoder objects; when applied to a transcoder returned by `make-transcoder`, they return the *codec*, *eol-style*, *handling-mode* arguments. When applied to the binary transcoder, they return `#f`.

`(bytevector->string` *bytevector transcoder*`)` procedure

Returns the string that results from transcoding the *bytevector* according to the input direction of the *transcoder*.

`(string->bytevector` *string transcoder*`)` procedure

Returns the bytevector that results from transcoding the *string* according to the output direction of the *transcoder*.

7.2.5. End of file object

The end of file object is returned by various I/O procedures when they reach end of file.

`(eof-object)` procedure

Returns the end of file object.

```
(eqv? (eof-object) (eof-object))
          ⟹  #t(eq? (eof-object) (eof-object))
          ⟹  #t
```

Note: The end of file object is not a datum value, and thus has no external representation.

`(eof-object?` *obj*`)` procedure

Returns `#t` if *obj* is the end of file object, otherwise returns `#f`.

7.2.6. Input and output ports

The operations described in this section are common to input and output ports, both binary and textual. Every port is associated with a transcoder; the transcoder of a binary port is the binary transcoder, and the transcoder of a textual port is not the binary transcoder. A port may also have an associated *position* that specifies a particular place within its data sink or source as a byte count from the beginning of the sink or source, and may also provide operations for inspecting and setting that place. (Ends of file do not count as bytes.)

`(port?` *obj*`)` procedure

Returns `#t` if the argument is a port, and returns `#f` otherwise.

`(port-transcoder` *port*`)` procedure

Returns the transcoder associated with *port*.

`(binary-port?` *port*`)` procedure

Returns `#t` if the transcoder associated with the *port* is the binary transducer, and returns `#f` otherwise.

`(transcoded-port` *binary-port transcoder*`)` procedure

Transcoder must not be the binary transcoder. The `transcoded-port` procedure returns a new textual port with the specified *transcoder*. Otherwise the new textual port's state is largely the same as that of the *binary-port*. If the *binary-port* is an input port, then the new textual port will be an input port and will transcode the bytes that have not yet been read from the *binary-port*. If the

binary-port is an output port, then the new textual port will be an output port and will transcode output characters into bytes that are written to the byte sink represented by the *binary-port*.

As a side effect, however, the `transcoded-port` procedure closes *binary-port* in a special way that allows the new textual port to continue to use the byte source or sink represented by the *binary-port*, even though the *binary-port* itself is closed and cannot be used by the input and output operations described in this chapter.

Rationale: Closing the *binary-port* precludes interference between the *binary-port* and the textual port constructed from it.

`(port-has-port-position?` *port*`)` procedure
`(port-position` *port*`)` procedure

The `port-has-port-position?` procedure returns `#t` if the port supports the `port-position` operation, and `#f` otherwise.

The `port-position` procedure returns the exact non-negative integer index of the position at which the next byte would be read from or written to the port. This procedure raises an exception with condition type `&assertion` if the port does not support the operation.

`(port-has-set-port-position!?` *port*`)` procedure
`(set-port-position!` *port pos*`)` procedure

Pos must be a non-negative exact integer.

The `port-has-set-port-position?` procedure returns `#t` if the port supports the `set-port-position!` operation, and `#f` otherwise.

The `set-port-position!` procedure sets the current byte position of the port to *pos*. If *port* is an output or combined input and output port, this first flushes *port*. (See `flush-output-port`, section 7.2.10.) This procedure raises an exception with condition type `&assertion` if the port does not support the operation.

`(close-port` *port*`)` procedure

Closes the port, rendering the port incapable of delivering or accepting data. If *port* is an output port, it is flushed before being closed. This has no effect if the port has already been closed. A closed port is still a port. The unspecified value is returned.

`(call-with-port` *port proc*`)` procedure

Proc must be a procedure that accepts a single argument. The `call-with-port` procedure calls *proc* with *port* as an argument. If *proc* returns, then the *port* is closed automatically and the values returned by *proc* are returned. If *proc* does not return, then the port is not closed automatically, except perhaps when it is possible to prove that the port will never again be used for a `lookahead`, `get`, or `put` operation.

7.2.7. Input ports

An input port allows reading an infinite sequence of bytes or characters punctuated by end of file objects. An input port connected to a finite data source ends in an infinite sequence of end of file objects.

It is unspecified whether a character encoding consisting of several bytes may have an end of file between the bytes. If, for example, `get-char` raises an `&i/o-decoding` exception because the character encoding at the port's position is incomplete up to the next end of file, a subsequent call to `get-char` may successfully decode a character if bytes completing the encoding are available after the end of file.

`(input-port?` *obj*`)` procedure

Returns `#t` if the argument is an input port (or a combined input and output port), and returns `#f` otherwise.

`(port-eof?` *input-port*`)` procedure

Returns `#t` if the `lookahead-u8` procedure (if *input-port* is a binary port) or the `lookahead-char` procedure (if *input-port* is a textual port) would return the end-of-file object, and returns `#f` otherwise.

`(open-file-input-port` *filename*`)` procedure
`(open-file-input-port` *filename file-options*`)` procedure
`(open-file-input-port` *filename file-options transcoder*`)` procedure

Returns an input port for the named file. The *file-options* and *transcoder* arguments are optional.

The *file-options* argument, which may determine various aspects of the returned port (see section 7.2.2), defaults to `(file-options)`.

If *transcoder* is specified, it becomes the transcoder associated with the returned port. If no *transcoder* is specified, then an implementation-dependent (and possibly locale-dependent) transcoder is associated with the port.

If the binary transcoder is passed as an explicit argument, then the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations will be implementation-dependent (and possibly transcoder-dependent).

Rationale: The byte position of a complexly transcoded port may not be well-defined, and may be hard to calculate even when defined, especially when transcoding is buffered.

`(open-bytevector-input-port` *bytevector*`)` procedure
`(open-bytevector-input-port` *bytevector transcoder*`)` procedure

Returns an input port whose bytes are drawn from the *bytevector*. If *transcoder* is specified, it becomes the transcoder associated with the returned port.

If no *transcoder* argument is given, or if the binary transcoder is passed as an explicit argument, then the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations will be implementation-dependent (and possibly transcoder-dependent).

If *bytevector* is modified after `open-bytevector-input-port` has been called, the effect on the returned port is unspecified.

`(open-string-input-port` *string*`)` procedure

Returns a textual input port whose characters are drawn from *string*. The port's transcoder is implementation-dependent, but is not the binary transcoder. Whether the port supports the `port-position` and `set-port-position!` operations is implementation-dependent.

If *string* is modified after `open-string-input-port` has been called, the effect on the returned port is unspecified.

`(standard-input-port)` procedure

Returns an input port connected to standard input, possibly a fresh one on each call. Whether the port is binary or textual is implementation-dependent, as is whether the port supports the `port-position` and `set-port-position!` operations.

Note: Implementations are encouraged to return a textual port when appropriate, and to associate an appropriate transcoder with the port.

`(make-custom-binary-input-port` *id read! get-position set-position! close*`)` procedure

Returns a newly created binary input port whose byte source is an arbitrary algorithm represented by the *read!* procedure. *Id* must be a string naming the new port, provided for informational purposes only. *Read!* must be a procedure, and should behave as specified below; it will be called by operations that perform binary input.

Each of the remaining arguments may be `#f`; if any of those arguments is not `#f`, it must be a procedure and should behave as specified below.

- `(`*read! bytevector start count*`)`

  Start will be a non-negative exact integer, *count* will be a positive exact integer, and *bytevector* will be a bytevector whose length is at least *start* + *count*. The *read!* procedure obtains up to *count* bytes from the byte source and writes those bytes into *bytevector* starting at index *start*. The *read!* procedure must return the number of bytes that it writes, as an exact integer. To indicate an end of file condition, the *read!* should write no bytes and return 0.

- `(`*get-position*`)`

  The *get-position* procedure (if supplied) returns the current position of the input port. If not supplied, the custom port will not support the `port-position` operation.

- `(`*set-position! k*`)`

  The *set-position!* procedure (if supplied) sets the position of the input port to *k*. If not supplied, the custom port will not support the `set-port-position!` operation.

- `(`*close*`)`

  The *close* procedure (if supplied) performs any actions that are necessary when the input port is closed.

7.2.8. Binary input

`(get-u8` *binary-input-port*`)` procedure

Reads from *binary-input-port*, blocking as necessary, until data are available from *binary-input-port* or until an end of file is reached. If a byte becomes available, `get-u8` returns the byte as an octet, and updates *binary-input-port* to point just past that byte. If no input byte is seen before an end of file is reached, then the end-of-file object is returned.

`(lookahead-u8` *binary-input-port*`)` procedure

The `lookahead-u8` procedure is like `get-u8`, but it does not update *binary-input-port* to point past the byte.

`(get-bytevector-n` *binary-input-port k*`)` procedure

Reads from *binary-input-port*, blocking as necessary, until *k* bytes are available from *binary-input-port* or until an end of file is reached. If *k* or more bytes are available before

an end of file, `get-bytevector-n` returns a bytevector of size k. If fewer bytes are available before an end of file, `get-bytevector-n` returns a bytevector containing those bytes. In either case, the input port is updated to point just past the bytes read. If an end of file is reached before any bytes are available, `get-bytevector-n` returns the end-of-file object.

`(get-bytevector-n!` *binary-input-port bytevector start count*`)` procedure

Count must be an exact, non-negative integer, specifying the number of bytes to be read. *bytevector* must be a bytevector with at least *start* + *count* elements.

The `get-bytevector-n!` procedure reads from *binary-input-port*, blocking as necessary, until *count* bytes are available from *binary-input-port* or until an end of file is reached. If *count* or more bytes are available before an end of file, they are written into *bytevector* starting at index *start*, and the result is *count*. If fewer bytes are available before the next end of file, the available bytes are written into *bytevector* starting at index *start*, and the result is the number of bytes actually read. In either case, the input port is updated to point just past the data read. If an end of file is reached before any bytes are available, `get-bytevector-n!` returns the end-of-file object.

`(get-bytevector-some` *binary-input-port*`)` procedure

Reads from *binary-input-port*, blocking as necessary, until data are available from *binary-input-port* or until an end of file is reached. If data become available, `get-bytevector-some` returns a freshly allocated bytevector of non-zero size containing the available data, and it updates *binary-input-port* to point just past that data. If no input bytes are seen before an end of file is reached, then the end-of-file object is returned.

`(get-bytevector-all` *binary-input-port*`)` procedure

Attempts to read all data until the next end of file, blocking as necessary. If one or more bytes are read, `get-bytevector-all` returns a bytevector containing all bytes up to the next end of file. Otherwise, `get-bytevector-all` returns the end-of-file object. Note that `get-bytevector-all` may block indefinitely, waiting to see an end of file, even though some bytes are available.

7.2.9. Textual input

`(get-char` *textual-input-port*`)` procedure

Reads from *textual-input-port*, blocking as necessary, until the complete encoding for a character is available from *textual-input-port*, or until the available input data cannot be the prefix of any valid encoding, or until an end of file is reached.

If a complete character is available before the next end of file, `get-char` returns that character, and updates the input port to point past the data that encoded that character. If an end of file is reached before any data are read, then `get-char` returns the end-of-file object.

`(lookahead-char` *textual-input-port*`)` procedure

The `lookahead-char` procedure is like `get-char`, but it does not update *textual-input-port* to point past the data that encode the character.

Note: With some of the standard transcoders described in this document, up to four bytes of lookahead are required. Nonstandard transcoders may require even more lookahead.

`(get-string-n` *textual-input-port k*`)` procedure

Reads from *textual-input-port*, blocking as necessary, until the encodings of k characters (including invalid encodings, if they don't raise an exception) are available, or until an end of file is reached.

If k or more characters are read before end of file, `get-string-n` returns a string consisting of those k characters. If fewer characters are available before an end of file, but one or more characters can be read, `get-string-n` returns a string containing those characters. In either case, the input port is updated to point just past the data read. If no data can be read before an end of file, then the end-of-file object is returned.

`(get-string-n!` *textual-input-port string start count*`)` procedure

Start and *count* must be an exact, non-negative integer, specifying the number of characters to be read. *string* must be a string with at least *start* + *count* characters.

Reads from *textual-input-port* in the same manner as `get-string-n`. If *count* or more characters are available before an end of file, they are written into string starting at index *start*, and *count* is returned. If fewer characters are available before an end of file, but one or more can be read, then those characters are written into string starting at index *start*, and the number of characters actually read is returned. If no characters can be read before an end of file, then the end-of-file object is returned.

`(get-string-all` *textual-input-port*`)` procedure

Reads from *textual-input-port* until an end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

If data are available before the end of file, a string containing all the text decoded from that data are returned. If no data precede the end of file, the end-of-file object file object is returned.

`(get-line` *textual-input-port*`)` procedure

Reads from *textual-input-port* up to and including the next end-of-line encoding or line separator character (U+2028) or end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

If an end-of-line encoding or line separator is read, then a string containing all of the text up to (but not including) the end-of-line encoding is returned, and the port is updated to point just past the end-of-line encoding or line separator. If an end of file is encountered before any end-of-line encoding is read, but some data have been read and decoded as characters, then a string containing those characters is returned. If an end of file is encountered before any data are read, then the end-of-file object is returned.

`(get-datum` *textual-input-port*`)` procedure

Reads an external representation from *textual-input-port* and returns the datum it represents. The `get-datum` procedure returns the next datum that can be parsed from the given *textual-input-port*, updating *textual-input-port* to point exactly past the end of the external representation of the object.

Any ⟨interlexeme space⟩ (see report section 3.2) in the input is first skipped. If an end of file occurs after the ⟨interlexeme space⟩, the end of file object (see section 7.2.5) is returned.

If a character inconsistent with an external representation is encountered in the input, an exception with condition types `&lexical` and `&i/o-read` is raised. Also, if the end of file is encountered after the beginning of an external representation, but the external representation is incomplete and therefore cannot be parsed, an exception with condition types `&lexical` and `&i/o-read` is raised.

7.2.10. Output ports

An output port is a sink to which bytes or characters are written. These data may control external devices, or may produce files and other objects that may subsequently be opened for input.

`(output-port?` *obj*`)` procedure

Returns `#t` if the argument is an output port (or a combined input and output port), and returns `#f` otherwise.

`(flush-output-port` *output-port*`)` procedure

Flushes any output from the buffer of *output-port* to the underlying file, device, or object. The unspecified value is returned.

`(output-port-buffer-mode` *output-port*`)` procedure

Returns the symbol that represents the buffer-mode of *output-port*.

`(open-file-output-port` *filename*`)` procedure
`(open-file-output-port` *filename file-options*`)` procedure
`(open-file-output-port` *filename file-options buffer-mode*`)` procedure
`(open-file-output-port` *filename file-options buffer-mode transcoder*`)` procedure

Returns an output port for the named file.

The *file-options* argument, which may determine various aspects of the returned port (see section 7.2.2), defaults to `(file-options)`.

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The *buffer-mode* argument defaults to `block`.

If *transcoder* is specified, it becomes the transcoder associated with the returned port. If no *transcoder* is specified, then an implementation-dependent (and possibly locale-dependent) transcoder is associated with the port.

If the binary transcoder is passed as an explicit argument, then the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations will be implementation-dependent (and possibly transcoder-dependent).

Rationale: The byte position of a complexly transcoded port may not be well-defined, and may be hard to calculate even when defined, especially when transcoding is buffered.

`(open-bytevector-output-port)` procedure
`(open-bytevector-output-port` *transcoder*`)` procedure

Returns two values: an output port and a procedure. The output port accumulates the data written to it for later extraction by the procedure.

If *transcoder* is specified, it becomes the transcoder associated with the port. If no *transcoder* argument is given, or if the binary transcoder is passed as an explicit argument, then the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!`

operations will be implementation-dependent (and possibly transcoder-dependent).

When the procedure is called without arguments, it returns a bytevector consisting of the port's accumulated data and removes the accumulated data from the port.

`(call-with-bytevector-output-port` *proc*`)` procedure
`(call-with-bytevector-output-port` *proc transcoder*`)` procedure

Proc must be a procedure accepting one argument. Creates an output port that accumulates the data written to it and calls *proc* with that output port as an argument. Whenever *proc* returns, a bytevector consisting of the port's accumulated data is returned and the accumulated data is removed from the port.

The transcoder associated with the output port is determined as for a call to `open-bytevector-output-port`.

`(open-string-output-port` *proc*`)` procedure

Returns two values: a textual output port and a procedure. The output port accumulates the characters written to it for later extraction by the procedure.

The port's transcoder is implementation-dependent, but is not the binary transcoder. Whether the port supports the `port-position` and `set-port-position!` operations is implementation-dependent.

When the procedure is called without arguments, it returns a string consisting of the port's accumulated characters and removes the accumulated characters from the port.

`(call-with-string-output-port` *proc*`)` procedure

Proc must be a procedure accepting one argument. Creates a textual output port that accumulates the characters written to it and calls *proc* with that output port as an argument. Whenever *proc* returns, a string consisting of the port's accumulated characters is returned and the accumulated characters are removed from the port.

The transcoder associated with the textual output port is implementation-dependent, but is not the binary transcoder. Whether the port supports the `port-position` and `set-port-position!` operations is implementation-dependent.

`(standard-output-port)` procedure
`(standard-error-port)` procedure

Returns an output port connected to the standard output or standard error, respectively, possibly a fresh one on each call. Whether the port is binary or textual is implementation-dependent, as is whether the port supports the `port-position` and `set-port-position!` operations.

Note: Implementations are encouraged to return a textual port when appropriate, and to associate an appropriate transcoder with the port.

`(make-custom-binary-output-port` *id write! get-position set-position! close*`)` procedure

Returns a newly created binary output port whose byte sink is an arbitrary algorithm represented by the *write!* procedure. *Id* must be a string naming the new port, provided for informational purposes only. *Write!* must be a procedure, and should behave as specified below; it will be called by operations that perform binary output.

Each of the remaining arguments may be `#f`; if any of those arguments is not `#f`, it must be a procedure and should behave as specified in the description of `make-custom-binary-input-port`.

- (*write! bytevector start count*)

  Start and *count* will be non-negative exact integers, and *bytevector* will be a bytevector whose length is at least *start* + *count*. The *write!* procedure reads up to *count* bytes from *bytevector* starting at index *start*, and forward them to the byte sink. If *count* is 0, then the *write!* procedure should have the effect of passing an end-of-file object to the byte sink. In any case, the *write!* procedure must return the number of bytes that it reads, as an exact integer.

7.2.11. Binary output

`(put-u8` *binary-output-port octet*`)` procedure

Writes *octet* to the output port and returns the unspecified value.

`(put-bytevector` *binary-output-port bytevector*`)` procedure
`(put-bytevector` *binary-output-port bytevector start*`)` procedure
`(put-bytevector` *binary-output-port bytevector start count*`)` procedure

Start and *count* must be non-negative exact integers that default to 0 and `(bytevector-length` *bytevector*`)` − *start*, respectively. *bytevector* must have a size of at least *start* + *count*. The `put-bytevector` procedure writes *count* bytes of the bytevector *bytevector*, starting at index *start*, to the output port. The unspecified value is returned.

7.2.12. Textual output

`(put-char` *textual-output-port char*`)` procedure

Writes *char* to the port. The unspecified value is returned.

`(put-string` *textual-output-port string*`)` procedure

Writes the characters of *string* to the port. The unspecified value is returned.

`(put-string-n` *textual-output-port string*`)` procedure
`(put-string-n` *textual-output-port string start*`)` procedure
`(put-string-n` *textual-output-port string start count*`)` procedure

Start and *count* must be non-negative exact integers. *String* must have a length of at least *start*+*count*. *Start* defaults to 0. *Count* defaults to `(string-length` *string*`)` − *start*. Writes the *count* characters of *string*, starting at index *start*, to the port. The unspecified value is returned.

`(put-datum` *textual-output-port datum*`)` procedure

Datum should be a datum value. The `put-datum` procedure writes an external representation of *datum* to *textual-output-port*. The specific external representation is implementation-dependent.

Note: The `put-datum` procedure merely writes the external representation, but no trailing delimiter. If `put-datum` is used to write several subsequent external representations to an output port, care must be taken to delimit them properly so they can be read back in by subsequent calls to `get-datum`.

7.2.13. Input/output ports

`(open-file-input/output-port` *filename*`)` procedure
`(open-file-input/output-port` *filename file-options*`)` procedure
`(open-file-input/output-port` *filename file-options buffer-mode*`)` procedure
`(open-file-input/output-port` *filename file-options buffer-mode transcoder*`)` procedure

Returns a single port that is both an input port and an output port for the named file. The optional arguments default as described in the specification of `open-file-output-port`. If the input/output port supports `port-position` and/or `set-port-position!`, then the same port position is used for both input and output.

`(make-custom-binary-input/output-port` *id read! write! get-position set-position! close*`)` procedure

Returns a newly created binary input/output port whose byte source and sink are arbitrary algorithms represented by the *read!* and *write!* procedures. *Id* must be a string naming the new port, provided for informational purposes only. *Read!* and *write!* must be procedures, and should behave as specified for the `make-custom-binary-input-port` and `make-custom-binary-output-port` procedures.

Each of the remaining arguments may be `#f`; if any of those arguments is not `#f`, it must be a procedure and should behave as specified in the description of `make-custom-binary-input-port`.

7.3. Simple I/O

This section describes the `(r6rs i/o simple)` library, which provides a somewhat more convenient interface for performing textual I/O on ports. This library implements most of the I/O procedures of the previous version of this report [5].

`(eof-object)` procedure
`(eof-object?` *obj*`)` procedure

These are the same as `eof-object` and `eof-object?` from the `(r6rs ports)` library.

`(call-with-input-file` *filename proc*`)` procedure
`(call-with-output-file` *filename proc*`)` procedure

Proc must be a procedure accepting a single argument. These procedures open the file named by *filename* for input or for output, with no specified file options, and call *proc* with the obtained port as an argument. If *proc* returns, then the port is closed automatically and the values returned by *proc* are returned. If *proc* does not return, then the port is not closed automatically, unless it is possible to prove that the port will never again be used for an I/O operation.

`(input-port?` *obj*`)` procedure
`(output-port?` *obj*`)` procedure

These are the same as the `input-port?` and `output-port?` procedures in the `(r6rs i/o ports)` library.

`(current-input-port)` procedure
`(current-output-port)` procedure

These return default ports for input and output. Normally, these default ports are associated with standard input and standard output, respectively, but can be dynamically re-assigned using the `with-input-from-file` and `with-output-to-file` procedures described below.

`(with-input-from-file` *filename thunk*`)` procedure
`(with-output-to-file` *filename thunk*`)` procedure

Thunk must be a procedure that takes no arguments. The file is opened for input or output using empty file options, and *thunk* is called with no arguments. During the dynamic extent of the call to *thunk*, the obtained port is made the value returned by `current-input-port` or `current-output-port` procedures; the previous default values are reinstated when the dynamic extent is exited. When *thunk* returns, the port is closed automatically, and the previous values for `current-input-port`. The values returned by *thunk* are returned. If an escape procedure is used to escape back into the call to *thunk* after *thunk* is returned, the behavior is unspecified.

`(open-input-file` *filename*`)` procedure

This opens *filename* for input, with empty file options, and returns the obtained port.

`(open-output-file` *filename*`)` procedure

This opens *filename* for output, with empty file options, and returns the obtained port.

`(close-input-port` *input-port*`)` procedure
`(close-output-port` *output-port*`)` procedure

This closes *input-port* or *output-port*, respectively.

`(read-char)` procedure
`(read-char` *textual-input-port*`)` procedure

This reads from *textual-input-port*, blocking as necessary until a character is available from *textual-input-port*, or the data that are available cannot be the prefix of any valid encoding, or an end of file is reached.

If a complete character is available before the next end of file, `read-char` returns that character, and updates the input port to point past that character. If an end of file is reached before any data are read, then `read-char` returns the end-of-file object.

If *textual-input-port* is omitted, it defaults to the value returned by `current-input-port`.

`(peek-char)` procedure
`(peek-char` *textual-input-port*`)` procedure

This is the same as `read-char`, but does not consume any data from the port.

`(read)` procedure
`(read` *textual-input-port*`)` procedure

Reads an external representation from *textual-input-port* and returns the datum it represents. The `read` procedure operates in the same way as `get-datum`, see section 7.2.9.

If *textual-input-port* is omitted, it defaults to the value returned by `current-input-port`.

`(write-char` *char*`)` procedure
`(write-char` *char textual-output-port*`)` procedure

Writes an encoding of the character *char* to the *textual-output-port*. The unspecified value is returned.

If *textual-output-port* is omitted, it defaults to the value returned by `current-output-port`.

`(newline)` procedure
`(newline` *textual-output-port*`)` procedure

This is equivalent to using `write-char` to write `#\linefeed` to *textual-output-port*.

If *textual-output-port* is omitted, it defaults to the value returned by `current-output-port`.

`(display` *obj*`)` procedure
`(display` *obj textual-output-port*`)` procedure

Writes a representation of *obj* to the given *textual-output-port*. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by `write-char` instead of by `write`. `display` returns the unspecified value. The *textual-output-port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

`(write` *obj*`)` procedure
`(write` *obj textual-output-port*`)` procedure

Writes the external representation of *obj* to *textual-output-port*. The `write` procedure operates in the same way as `put-datum`; see section 7.2.12.

If *textual-output-port* is omitted, it defaults to the value returned by `current-output-port`.

8. File system

This chapter describes the `(r6rs files)` library for operations on the file system. This library, in addition to the procedures described here, also exports the I/O condition types described in section 7.1.

`(file-exists?` *filename*`)` procedure

Filename must be a filename (see section 7.2.1). The `file-exists?` procedure returns `#t` if the named file exists at the time the procedure is called, `#f` otherwise.

`(delete-file` *filename*`)` procedure

Filename must be a filename (see section 7.2.1). The `delete-file` procedure deletes the named file if it exists and if it is possible, and returns the unspecified value. If the file does not exist or cannot be deleted, an exception with condition type `&i/o-filename` is raised.

9. Arithmetic

This chapter describes Scheme's libraries for more specialized numerical operations: fixnum and flonum arithmetic, as well as bitwise operations on exact integers.

9.1. Fixnums

Every implementation must define its fixnum range as a closed interval

$$[-2^{w-1}, 2^{w-1} - 1]$$

such that w is a (mathematical) integer $w \geq 24$. Every mathematical integer within an implementation's fixnum range must correspond to an exact integer that is representable within the implementation. A fixnum is an exact integer whose value lies within this fixnum range.

This section describes the `(r6rs arithmetic fx)` library, which defines various operations on fixnums. Fixnum operations perform integer arithmetic on their fixnum arguments, but raise an exception with condition type `&implementation-restriction` if the result is not a fixnum.

This section uses *fx*, fx_1 and fx_2 as parameter names for arguments that must be fixnums.

`(fixnum?` *obj*`)` procedure

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

`(fixnum-width)` procedure
`(least-fixnum)` procedure
`(greatest-fixnum)` procedure

These procedures return w, -2^{w-1} and $2^{w-1} - 1$: the width, minimum and the maximum value of the fixnum range, respectively.

`(fx=?` fx_1 fx_2 fx_3 ...`)` procedure
`(fx>?` fx_1 fx_2 fx_3 ...`)` procedure
`(fx<?` fx_1 fx_2 fx_3 ...`)` procedure
`(fx>=?` fx_1 fx_2 fx_3 ...`)` procedure
`(fx<=?` fx_1 fx_2 fx_3 ...`)` procedure

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, `#f` otherwise.

`(fxzero?` *fx*`)` procedure
`(fxpositive?` *fx*`)` procedure
`(fxnegative?` *fx*`)` procedure
`(fxodd?` *fx*`)` procedure
`(fxeven?` *fx*`)` procedure

These numerical predicates test a fixnum for a particular property, returning `#t` or `#f`. The five properties tested by these procedures are: whether the number is zero, greater than zero, less than zero, odd, or even.

`(fxmax` fx_1 fx_2 ...`)` procedure
`(fxmin` fx_1 fx_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

`(fx+` fx_1 fx_2`)` procedure
`(fx*` fx_1 fx_2`)` procedure

These procedures return the sum or product of their arguments, provided that sum or product is a fixnum. An exception with condition type `&implementation-restriction` is raised if that sum or product is not a fixnum.

Rationale: These procedures are restricted to two arguments because their generalizations to three or more arguments would require precision proportional to the number of arguments.

`(fx-` fx_1 fx_2`)` procedure
`(fx-` *fx*`)` procedure

With two arguments, this procedure returns the difference of its arguments, provided that difference is a fixnum.

With one argument, this procedure returns the additive inverse of its argument, provided that integer is a fixnum.

An exception with condition type `&assertion` is raised if the mathematically correct result of this procedure is not a fixnum.

```
(fx- (least-fixnum))
          ⟹ &assertion exception
```

`(fxdiv-and-mod` fx_1 fx_2`)` procedure
`(fxdiv` fx_1 fx_2`)` procedure
`(fxmod` fx_1 fx_2`)` procedure
`(fxdiv0-and-mod0` fx_1 fx_2`)` procedure
`(fxdiv0` fx_1 fx_2`)` procedure
`(fxmod0` fx_1 fx_2`)` procedure

Fx_2 must be nonzero. These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section 9.9.3.

```
(fxdiv fx1 fx2)                  ⟹ fx1 div fx2
(fxmod fx1 fx2)                  ⟹ fx1 mod fx2
(fxdiv-and-mod fx1 fx2)
          ⟹ fx1 div fx2, fx1 mod fx2
                                 ; two return values
(fxdiv0 fx1 fx2)                 ⟹ fx1 div0 fx2
(fxmod0 fx1 fx2)                 ⟹ fx1 mod0 fx2
(fxdiv0-and-mod0 fx1 fx2)
          ⟹ fx1 fx1 div0 fx2, fx1 mod0 fx2
                                 ; two return values
```

`(fx+/carry` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Returns the two fixnum results of the following computation:

```
(let* ((s (+ fx1 fx2 fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`(fx-/carry` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Returns the two fixnum results of the following computation:

```
(let* ((d (- fx1 fx2 fx3))
       (d0 (mod0 d (expt 2 (fixnum-width))))
       (d1 (div0 d (expt 2 (fixnum-width)))))
  (values d0 d1))
```

`(fx*/carry` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Returns the two fixnum results of the following computation:

```
(let* ((s (+ (* fx1 fx2) fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`(fxnot` *fx*`)` procedure

Returns the unique fixnum that is congruent mod 2^w to the one's-complement of *fx*.

`(fxand` *fx*$_1$ ...`)` procedure
`(fxior` *fx*$_1$ ...`)` procedure
`(fxxor` *fx*$_1$ ...`)` procedure

These procedures return the fixnum that is the bit-wise "and", "inclusive or", or "exclusive or" of the two's complement representations of their arguments. If they are passed only one argument, they return that argument. If they are passed no arguments, they return the fixnum (either -1 or 0) that acts as identity for the operation.

`(fxif` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Returns the fixnum result of the following computation:

```
(fxior (fxand fx1 fx2)
       (fxand (fxnot fx1) fx3))
```

`(fxbit-count` *fx*`)` procedure

If *fx* is non-negative, this procedure returns the number of 1 bits in the two's complement representation of *fx*. Otherwise it returns the number of 0 bits in the two's complement representation of *fx*.

`(fxlength` *fx*`)` procedure

Returns the fixnum result of the following computation:

```
(do ((result 0 (+ result 1))
     (bits (if (fxnegative? fx)
               (fxnot fx)
               fx)
           (fxlogical-shift-right bits 1)))
    ((fxzero? bits)
     result))
```

`(fxfirst-bit-set` *fx*`)` procedure

Returns the index of the least significant 1 bit in the two's complement representation of *fx*. If *fx* is 0, then -1 is returned.

```
(fxfirst-bit-set 0)         ⟹ -1
(fxfirst-bit-set 1)         ⟹ 0
(fxfirst-bit-set -4)        ⟹ 2
```

`(fxbit-set?` *fx* *fx*$_2$`)` procedure

Fx$_2$ must be non-negative and less than `(fixnum-width)`. The `fxbit-set?` procedure returns the fixnum result of the following computation:

```
(not
  (fxzero?
    (fxand fx1
           (fxlogical-shift-left 1 fx2))))
```

`(fxcopy-bit` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Fx$_2$ must be non-negative and less than `(fixnum-width)`. *Fx*$_3$ must be 0 or 1. The `fxbit-field` procedure returns the fixnum result of the following computation:

```
(let* ((mask (fxnot
              (fxlogical-shift-left -1 fx3))))
  (fxlogical-shift-right (fxand fx1 mask)
                         fx2))
```

`(fxbit-field` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Fx$_2$ and *fx*$_3$ must be non-negative and less than `(fixnum-width)`. Moreover, *fx*$_2$ must be less than or equal to *fx*$_3$. The `fxbit-field` procedure returns the fixnum result of the following computation:

```
(let* ((mask (fxnot
              (fxlogical-shift-left -1 fx3))))
  (fxlogical-shift-right (fxand fx1 mask)
                         fx2))
```

`(fxcopy-bit-field` fx_1 fx_2 fx_3 fx_4`)` procedure

Fx_2 and fx_3 must be non-negative and less than `(fixnum-width)`. Moreover, fx_2 must be less than or equal to fx_3. The `fxcopy-bit-field` procedure returns the fixnum result of the following computation:

```
(let* ((to    fx1)
       (start fx2)
       (end   fx3)
       (from  fx4)
       (mask1 (fxlogical-shift-left -1 start))
       (mask2 (fxnot
               (fxlogical-shift-left -1 end)))
       (mask (fxand mask1 mask2)))
  (fxif mask
        (fxlogical-shift-left from start)
        to))
```

`(fxarithmetic-shift` fx_1 fx_2`)` procedure

The absolute value of the fx_2 must be less than `(fixnum-width)`. If

```
(* fx1 (expt 2 fx2))
```

is a fixnum, then that fixnum is returned. Otherwise an exception with condition type `&implementation-restriction` is raised.

`(fxarithmetic-shift-left` fx_1 fx_2`)` procedure
`(fxarithmetic-shift-right` fx_1 fx_2`)` procedure

Fx_2 must be non-negative. `fxarithmetic-shift-left` behaves the same as `fxarithmetic-shift`, and `(fxarithmetic-shift-right` fx_1 fx_2`)` behaves the same as `(fxarithmetic-shift` fx_1 `(fixnum-` fx_2`))`.

`(fxrotate-bit-field` fx_1 fx_2 fx_3 fx_4`)` procedure

Fx_2, fx_3, and fx_4 must be non-negative and less than `(fixnum-width)`. Fx_4 must be less than the difference between fx_3 and fx_3. The `fxrotate-bit-field` procedure returns the result of the following computation:

```
(let* ((n     fx1)
       (start fx2)
       (end   fx3)
       (count fx4)
       (width (fx- end start)))
  (if (fxpositive? width)
      (let* ((count (fxmod count width))
             (field0
               (fxbit-field n start end))
             (field1
               (fxlogical-shift-left
                 field0 count))
             (field2
               (fxlogical-shift-right
                 field0 (fx- width count)))
             (field (fxior field1 field2)))
        (fxcopy-bit-field n start end field))
      n))
```

`(fxreverse-bit-field` fx_1 fx_2 fx_3`)` procedure

Fx_2 and fx_3 must be non-negative and less than `(fixnum-width)`. Moreover, fx_2 must be less than or equal to fx_3. The `fxreverse-bit-field` procedure returns the fixnum obtained from fx_1 by reversing the bit field specified by fx_2 and fx_3.

```
(fxreverse-bit-field #b1010010 1 4)
          ⟹  88 ; #b1011000
(fxreverse-bit-field #b1010010 91 -4)
          ⟹  82 ; #b1010010
```

9.2. Flonums

This section describes the `(r6rs arithmetic flonum)` library.

This section uses fl, fl_1 and fl_2 as parameter names for arguments that must be flonums, and ifl, ifl_1 and ifl_2 as parameter names for arguments that must be integer-valued flonums, i.e. flonums for which the `integer-valued?` predicate returns true.

`(flonum?` *obj*`)` procedure

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

`(fl=?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl<?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl<=?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl>?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl>=?` fl_1 fl_2 fl_3 ...`)` procedure

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, `#f` otherwise. These predicates are required to be transitive.

```
(fl= +inf.0 +inf.0)          ⟹  #t
(fl= -inf.0 +inf.0)          ⟹  #f
(fl= -inf.0 -inf.0)          ⟹  #t
(fl= 0.0 -0.0)               ⟹  #t
(fl< 0.0 -0.0)               ⟹  #f
(fl= +nan.0 fl)              ⟹  #f
(fl< +nan.0 fl)              ⟹  #f
```

`(flinteger?` *fl*`)` procedure
`(flzero?` *fl*`)` procedure
`(flpositive?` *fl*`)` procedure
`(flnegative?` *fl*`)` procedure
`(flodd?` *ifl*`)` procedure
`(fleven?` *ifl*`)` procedure
`(flfinite?` *fl*`)` procedure
`(flinfinite?` *fl*`)` procedure
`(flnan?` *fl*`)` procedure

These numerical predicates test a flonum for a particular property, returning `#t` or `#f`. The `flinteger?` procedure tests whether the number is an integer, `flzero?` tests whether it is `fl=?` to zero, `flpositive?` tests whether it is greater than zero, `flnegative?` tests whether it is less than zero, `flodd?` tests whether it is odd, `fleven?` tests whether it is even, `flfinite?` tests whether it is not an infinity and not a NaN, `flinfinite?` tests whether it is an infinity, and `flnan?` tests whether it is a NaN.

```
(flnegative? -0.0)          ⟹ #f
(flfinite? +inf.0)          ⟹ #f
(flfinite? 5.0)             ⟹ #t
(flinfinite? 5.0)           ⟹ #f
(flinfinite? +inf.0)        ⟹ #t
```

Note: `(flnegative? -0.0)` must return `#f`, else it would lose the correspondence with `(fl< -0.0 0.0)`, which is `#f` according to the IEEE standards.

`(flmax` fl_1 fl_2 ...`)` procedure
`(flmin` fl_1 fl_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

`(fl+` fl_1 ...`)` procedure
`(fl*` fl_1 ...`)` procedure

These procedures return the flonum sum or product of their flonum arguments. In general, they should return the flonum that best approximates the mathematical sum or product. (For implementations that represent flonums as IEEE binary floating point numbers, the meaning of "best" is defined by the IEEE standards.)

```
(fl+ +inf.0 -inf.0)         ⟹ +nan.0
(fl+ +nan.0 fl)             ⟹ +nan.0
(fl* +nan.0 fl)             ⟹ +nan.0
```

`(fl-` fl_1 fl_2 ...`)` procedure
`(fl-` *fl*`)` procedure
`(fl/` fl_1 fl_2 ...`)` procedure
`(fl/` *fl*`)` procedure

With two or more arguments, these procedures return the flonum difference or quotient of their flonum arguments, associating to the left. With one argument, however, they return the additive or multiplicative flonum inverse of their argument. In general, they should return the flonum that best approximates the mathematical difference or quotient. (For implementations that represent flonums as IEEE binary floating point numbers, the meaning of "best" is reasonably well-defined by the IEEE standards.)

```
(fl- +inf.0 +inf.0)         ⟹ +nan.0
```

For undefined quotients, `fl/` behaves as specified by the IEEE standards:

```
(fl/ 1.0 0.0)               ⟹ +inf.0
(fl/ -1.0 0.0)              ⟹ -inf.0
(fl/ 0.0 0.0)               ⟹ +nan.0
```

`(flabs` *fl*`)` procedure

Returns the absolute value of *fl*.

`(fldiv-and-mod` fl_1 fl_2`)` procedure
`(fldiv` fl_1 fl_2`)` procedure
`(flmod` fl_1 fl_2`)` procedure
`(fldiv0-and-mod0` fl_1 fl_2`)` procedure
`(fldiv0` fl_1 fl_2`)` procedure
`(flmod0` fl_1 fl_2`)` procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section 9.9.3. For zero divisors, these procedures may return a NaN or some meaningless flonum.

```
(fldiv fl1 fl2)             ⟹ fl1 div fl2
(flmod fl1 fl2)             ⟹ fl1 mod fl2
(fldiv-and-mod fl1 fl2)
        ⟹ fl1 div fl2, fl1 mod fl2
                            ; two return values
(fldiv0 fl1 fl2)            ⟹ fl1 div0 fl2
(flmod0 fl1 fl2)            ⟹ fl1 mod0 fl2
(fldiv0-and-mod0 fl1 fl2)
        ⟹ fl1 div0 fl2, fl1 mod0 fl2
                            ; two return values
```

`(flnumerator` *fl*`)` procedure
`(fldenominator` *fl*`)` procedure

These procedures return the numerator or denominator of *fl* as a flonum; the result is computed as if *fl* was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0.0 is defined to be 1.0.

```
(flnumerator +inf.0)        ⟹ +inf.0
(flnumerator -inf.0)        ⟹ -inf.0
(fldenominator +inf.0)      ⟹ 1.0
(fldenominator -inf.0)      ⟹ 1.0
(flnumerator 0.75)          ⟹ 3.0 ; example
(fldenominator 0.75)        ⟹ 4.0 ; example
```

The following behavior is strongly recommended but not required:

```
(flnumerator -0.0)          ⟹ -0.0
```

`(flfloor` *fl*`)` procedure
`(flceiling` *fl*`)` procedure
`(fltruncate` *fl*`)` procedure
`(flround` *fl*`)` procedure

These procedures return integral flonums for flonum arguments that are not infinities or NaNs. For such arguments, `flfloor` returns the largest integral flonum not larger than *fl*. The `flceiling` procedure returns the smallest integral flonum not smaller than *fl*. The `fltruncate` procedure returns the integral flonum closest to *fl* whose absolute value is not larger than the absolute value of *fl*. The `flround` procedure returns the closest integral flonum to *fl*, rounding to even when *fl* is halfway between two integers.

Rationale: The `flround` procedure rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

Although infinities and NaNs are not integers, these procedures return an infinity when given an infinity as an argument, and a NaN when given a NaN:

```
(flfloor +inf.0)            ⟹ +inf.0
(flceiling -inf.0)          ⟹ -inf.0
(fltruncate +nan.0)         ⟹ +nan.0
```

`(flexp` *fl*`)` procedure
`(fllog` *fl*`)` procedure
`(fllog` fl_1 fl_2`)` procedure
`(flsin` *fl*`)` procedure
`(flcos` *fl*`)` procedure
`(fltan` *fl*`)` procedure
`(flasin` *fl*`)` procedure
`(flacos` *fl*`)` procedure
`(flatan` *fl*`)` procedure
`(flatan` fl_1 fl_2`)` procedure

These procedures compute the usual transcendental functions. The `flexp` procedure computes the base-e exponential of *fl*. The `fllog` procedure with a single argument computes the natural logarithm of *fl* (not the base ten logarithm); `(fllog` fl_1 fl_2`)` computes the base-fl_2 logarithm of fl_1. The `flasin`, `flacos`, and `flatan` procedures compute arcsine, arccosine, and arctangent, respectively. `(flatan` fl_1 fl_2`)` computes the arc tangent of fl_1/fl_2.

See report section 9.9.3 for the underlying mathematical operations. In the event that these operations do not yield a real result for the given arguments, the result may be a NaN, or may be some meaningless flonum.

Implementations that use IEEE binary floating point arithmetic are encouraged to follow the relevant standards for these procedures.

```
(flexp +inf.0)              ⟹ +inf.0
(flexp -inf.0)              ⟹ 0.0
(fllog +inf.0)              ⟹ +inf.0
(fllog 0.0)                 ⟹ -inf.0
(fllog -0.0)                ⟹ unspecified
        ; if -0.0 is distinguished
(fllog -inf.0)              ⟹ +nan.0
(flatan -inf.0)
        ⟹ -1.5707963267948965
        ; approximately
(flatan +inf.0)
        ⟹ 1.5707963267948965
        ; approximately
```

`(flsqrt` *fl*`)` procedure

Returns the principal square root of *fl*. For a negative argument, the result may be a NaN, or may be some meaningless flonum.

```
(flsqrt +inf.0)             ⟹ +inf.0
```

`(flexpt` fl_1 fl_2`)` procedure

Returns fl_1 raised to the power fl_2. fl_1 should be nonnegative; if fl_1 is negative, then the result may be a NaN, or may be some meaningless flonum. If fl_1 is zero, then the result is zero. For positive fl_1,

$$fl_1^{fl_2} = e^{fl_2 \log fl_1}$$

`&no-infinities` condition type
`(no-infinities?` *obj*`)` procedure
`&no-nans` condition type
`(no-nans?` *obj*`)` procedure

These condition types could be defined by the following code:

```
(define-condition-type &no-infinities
    &implementation-restriction
  no-infinities?)

(define-condition-type &no-nans
    &implementation-restriction
  no-nans?)
```

These types describe that a program has executed an arithmetic operations that is specified to return an infinity or a NaN, respectively, on a Scheme implementation that is not able to represent the infinity or NaN. (See report section 9.9.2.)

`(fixnum->flonum` *fx*`)` procedure

Returns a flonum that is numerically closest to *fx*.

Note: The result of this procedure may not be numerically equal to *fx*, because the fixnum precision may be greater than the flonum precision.

9.3. Exact bitwise arithmetic

This section describes the `(r6rs arithmetic bitwise)` library. The exact bitwise arithmetic provides generic operations on exact integers. This section uses *ei*, ei_1, ei_2, and ei_3 as parameter names that must be exact integers.

`(bitwise-not` *ei*`)` procedure

Returns the exact integer whose two's complement representation is the one's complement of the two's complement representation of *ei*.

`(bitwise-and` ei_1 ...`)` procedure
`(bitwise-ior` ei_1 ...`)` procedure
`(bitwise-xor` ei_1 ...`)` procedure

These procedures return the exact integer that is the bitwise "and", "inclusive or", or "exclusive or" of the two's complement representations of their arguments. If they are passed only one argument, they return that argument. If they are passed no arguments, they return the integer (either -1 or 0) that acts as identity for the operation.

`(bitwise-if` ei_1 ei_2 ei_3`)` procedure

Returns the exact integer that is the result of the following computation:

```
(bitwise-ior (bitwise-and ei1 ei2)
             (bitwise-and (bitwise-not ei1) ei3))
```

`(bitwise-bit-count` *ei*`)` procedure

If *ei* is non-negative, this procedure returns the number of 1 bits in the two's complement representation of *ei*. Otherwise it returns the number of 0 bits in the two's complement representation of *ei*.

`(bitwise-length` *ei*`)` procedure

These procedures return the exact integer that is the result of the following computation:

```
(do ((result 0 (+ result 1))
     (bits (if (bitwise-negative? ei)
               (bitwise-not ei)
               ei)
           (bitwise-arithmetic-shift bits -1)))
    ((zero? bits)
     result))
```

`(bitwise-first-bit-set` *ei*`)` procedure

Returns the index of the least significant 1 bit in the two's complement representation of *ei*. If *ei* is 0, then -1 is returned.

```
(bitwise-first-bit-set 0)   ⟹  -1
(bitwise-first-bit-set 1)   ⟹  0
(bitwise-first-bit-set -4)  ⟹  2
```

`(bitwise-bit-set?` ei_1 ei_2`)` procedure

Ei_2 must be non-negative.

Otherwise returns the result of the following computation:

```
(not (bitwise-zero?
      (bitwise-and
       (bitwise-arithmetic-shift-left 1 ei2)
       ei1)))
```

`(bitwise-copy-bit` ei_1 ei_2 ei_3`)` procedure

Ei_2 must be non-negative, and ei_3 must be either 0 or 1. The `bitwise-copy-bit` procedure returns the result of the following computation:

```
(let* ((mask (bitwise-arithmetic-shift-left 1 ei2)))
  (bitwise-if mask
              (bitwise-arithmetic-shift-left ei3 ei2)
              ei1))
```

`(bitwise-bit-field` ei_1 ei_2 ei_3`)` procedure

Ei_2 and ei_3 must be non-negative, and ei_2 must be less than or equal to ei_3. This procedure returns the result of the following computation:

```
(let* ((mask
        (bitwise-not
         (bitwise-arithmetic-shift-left -1 ei3))))
  (bitwise-arithmetic-shift-right
   (bitwise-and ei1 mask)
   ei2))
```

`(bitwise-copy-bit-field` ei_1 ei_2 ei_3 ei_4`)` procedure

Ei_2 and ei_3 must be non-negative, and ei_2 must be less than or equal to ei_3. The `bitwise-copy-bit-field` procedure returns the result of the following computation:

```
(let* ((to    ei1)
       (start ei2)
       (end   ei3)
       (from  ei4)
       (mask1
         (bitwise-arithmetic-shift-left -1 start))
       (mask2
         (bitwise-not
           (bitwise-arithmetic-shift-left -1 end)))
       (mask (bitwise-and mask1 mask2)))
  (bitwise-if mask
              (bitwise-arithmetic-shift-left from
                                             start)
              to))
```

`(bitwise-arithmetic-shift` ei_1 ei_2`)` procedure

Returns the result of the following computation:

```
(* ei1 (expt 2 ei2))
```

`(bitwise-arithmetic-shift-left` ei_1 ei_2`)` procedure
`(bitwise-arithmetic-shift-right` ei_1 ei_2`)` procedure

Ei_2 must be non-negative. The `bitwise-arithmetic-shift-left` procedure returns the same result as `bitwise-arithmetic-shift`, and `(bitwise-arithmetic-shift-right` ei_1 ei_2`)` returns the same result as `(bitwise-arithmetic-shift` ei_1 `(-` ei_2`))`.

`(bitwise-rotate-bit-field` ei_1 ei_2 ei_3 ei_4`)` procedure

Ei_2, ei_3, ei_4 must be non-negative, and ei_4 must be less than the difference between ei_2 and ei_3. The procedure returns the result of the following computation:

```
(let* ((n     ei1)
       (start ei2)
       (end   ei3)
       (count ei4)
       (width (- end start)))
  (if (positive? width)
      (let* ((count (mod count width))
             (field0
               (bitwise-bit-field n start end))
             (field1 (bitwise-arithmetic-shift-left
                       field0 count))
             (field2 (bitwise-arithmetic-shift-right
                       field0
                       (- width count)))
             (field (bitwise-ior field1 field2)))
        (bitwise-copy-bit-field n start end field))
      n))
```

`(bitwise-reverse-bit-field` ei_1 ei_2 ei_3`)` procedure

Ei_2 and ei_3 must be non-negative, and ei_2 must be less than or equal to ei_3. The `bitwise-reverse-bit-field` procedure returns the result obtained from the ei_1 by reversing the bit field specified by ei_2 and ei_3.

```
(bitwise-reverse-bit-field #b1010010 1 4)
          ⟹  88 ; #b1011000
(bitwise-reverse-bit-field #1010010 91 -4)
          ⟹  &assertion exception
```

10. syntax-case

The `(r6rs syntax-case)` library provides support for writing low-level macros in a high-level style, with automatic syntax checking, input destructuring, output restructuring, maintenance of lexical scoping and referential transparency (hygiene), and support for controlled identifier capture.

Rationale: While many syntax transformers are succinctly expressed using the high-level `syntax-rules` form, others are difficult or impossible to write, including some that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form, ones that maintain state or read from the file system, and ones that construct new identifiers. The `syntax-case` system [4] described here allows the programmer to write transformers that perform these sorts of transformations, and arbitrary additional transformations, without sacrificing the default enforcement of hygiene or the high-level pattern-based syntax matching and template-based output construction provided by `syntax-rules` (report section 9.21).

Because `syntax-case` does not require literals, including quoted lists or vectors, to be copied or even traversed, it may be able to preserve sharing and cycles within and among the constants of a program. It also supports source-object correlation, i.e., the maintenance of ties between the original source code and expanded output, allowing implementations to provide source-level support for debuggers and other tools.

10.1. Hygiene

Barendregt's *hygiene condition* [1] for the lambda-calculus is an informal notion that requires the free variables of an expression N that is to be substituted into another expression M not to be captured by bindings in M when such capture is not intended. Kohlbecker, et al [8] propose a corresponding *hygiene condition for macro expansion* that applies in all situations where capturing is not explicit: "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step". In the terminology of this document, the "generated identifiers" are those introduced by a transformer rather than those present in the form passed to the transformer, and a "macro transcription step" corresponds to a single call by the expander to a transformer. Also, the hygiene condition applies to all introduced bindings rather than to introduced variable bindings alone.

This leaves open what happens to an introduced identifier that appears outside the scope of a binding introduced by the same call. Such an identifier refers to the lexical binding in effect where it appears (within a `syntax` ⟨template⟩; see section 10.4) inside the transformer body or one of the helpers it calls. This is essentially the referential transparency property described by Clinger and Rees [3].

Thus, the hygiene condition can be restated as follows:

> A binding for an identifier introduced into the output of a transformer call from the expander must capture only references to the identifier introduced into the output of the same transformer call. A reference to an identifier introduced into the output of a transformer refers to the closest

enclosing binding for the introduced identifier or, if it appears outside of any enclosing binding for the introduced identifier, the closest enclosing lexical binding where the identifier appears (within a `syntax` ⟨template⟩) inside the transformer body or one of the helpers it calls.

Explicit captures are handled via `datum->syntax`; see section 10.6.

Operationally, the expander can maintain hygiene with the help of *marks* and *substitutions*. Marks are applied selectively by the expander to the output of each transformer it invokes, and substitutions are applied to the portions of each binding form that are supposed to be within the scope of the bound identifiers. Marks are used to distinguish like-named identifiers that are introduced at different times (either present in the source or introduced into the output of a particular transformer call), and substitutions are used to map identifiers to their expand-time values.

Each time the expander encounters a macro use, it applies an *antimark* to the input form, invokes the associated transformer, then applies a fresh mark to the output. Marks and antimarks cancel, so the portions of the input that appear in the output are effectively left unmarked, while the portions of the output that are introduced are marked with the fresh mark.

Each time the expander encounters a binding form it creates a set of substitutions, each mapping one of the (possibly marked) bound identifiers to information about the binding. (For a `lambda` expression, the expander might map each bound identifier to a representation of the formal parameter in the output of the expander. For a `let-syntax` form, the expander might map each bound identifier to the associated transformer.) These substitutions are applied to the portions of the input form in which the binding is supposed to be visible.

Marks and substitutions together form a *wrap* that is layered on the form being processed by the expander and pushed down toward the leaves as necessary. A wrapped form is referred to as a *wrapped syntax object*. Ultimately, the wrap may rest on a leaf that represents an identifier, in which case the wrapped syntax object is referred to more precisely as an *identifier*. An identifier contains a name along with the wrap. (Names are typically represented by symbols.)

When a substitution is created to map an identifier to an expand-time value, the substitution records the name of the identifier and the set of marks that have been applied to that identifier, along with the associated expand-time value. The expander resolves identifier references by looking for the latest matching substitution to be applied to the identifier, i.e., the outermost substitution in the wrap whose name and marks match the name and marks recorded in the substitution. The name matches if it is the same name (if using symbols, then by `eq?`), and the marks match if the marks recorded with the substitution are the same as those that appear *below* the substitution in the wrap, i.e., those that were applied *before* the substitution. Marks applied after a substitution, i.e., appear over the substitution in the wrap, are not relevant and are ignored.

An algebra that defines how marks and substitutions work more precisely is given in section 2.4 of Oscar Waddell's PhD thesis [11].

10.2. Syntax objects

A *syntax object* is a representation of a Scheme form that contains contextual information about the form in addition to its structure. This contextual information is used by the expander to maintain lexical scoping and may also be used by an implementation to maintain source-object correlation.

Syntax objects may be wrapped or unwrapped. A wrapped syntax object (section 10.1), consists of a *wrap* (section 10.1) and some internal representation of a Scheme form. (The internal representation is unspecified, but is typically a datum value or datum value annotated with source information.) A wrapped syntax object representing an identifier is itself referred to as an identifier; thus, the term *identifier* may refer either to the syntactic entity (symbol, variable, or keyword) or to the concrete representation of the syntactic entity as a syntax object. Wrapped syntax objects are distinct from other types of values.

An unwrapped syntax object is one that is unwrapped, fully or partially, i.e., whose outer layers consist of lists and vectors and whose leaves are either wrapped syntax objects or nonsymbol values.

The term syntax object is used in this document to refer to a syntax object that is either wrapped or unwrapped. More formally, a syntax object is:

- a pair or list of syntax objects,
- a vector of syntax objects,
- a nonlist, nonvector, nonsymbol value, or
- a wrapped syntax object.

The distinction between the terms "syntax object" and "wrapped syntax object" is important. For example, when invoked by the expander, a transformer (section 10.3) must accept a wrapped syntax object but may return any syntax object, including an unwrapped syntax object.

10.3. Transformers

In `define-syntax` (report section 9.3), `let-syntax`, and `letrec-syntax` forms (report section 9.20), a binding for a syntactic keyword must be an expression that evaluates to a transformer.

A transformer is a *transformation procedure* or a *variable transformer*. A transformation procedure is a procedure that must accept one argument, a wrapped syntax object (section 10.2) representing the input, and return a *syntax object* (section 10.2) representing the output. The procedure is called by the expander whenever a reference to a keyword with which it has been associated is found. If the keyword appears in the first position of a list-structured input form, the transformer receives the entire list-structured form, and its output replaces the entire form. If the keyword is found in any other definition or expression context, the transformer receives a wrapped syntax object representing just the keyword reference, and its output replaces just the reference. Except with variable transformers (see below), an exception with condition type `&syntax` is raised if the keyword appears on the left-hand side of a `set!` expression.

`(make-variable-transformer` *proc*`)` procedure

Proc must be a procedure that accepts one argument, a wrapped syntax object.

The `make-variable-transformer` procedure creates a *variable transformer*. A variable transformer is like an ordinary transformer except that, if a keyword associated with a variable transformer appears on the left-hand side of a `set!` expression, an exception is not raised. Instead, *proc* is called with a wrapped syntax object representing the entire `set!` expression as its argument, and its return value replaces the entire `set!` expression.

10.4. Parsing input and producing output

Transformers destructure their input with `syntax-case` and rebuild their output with `syntax`.

`(syntax-case` ⟨expression⟩ `(`⟨literal⟩ `...)` ⟨clause⟩ `...)` syntax

Syntax: Each ⟨literal⟩ must be an identifier. Each ⟨clause⟩ must take one of the following two forms.

`(`⟨pattern⟩ ⟨output expression⟩`)`
`(`⟨pattern⟩ ⟨fender⟩ ⟨output expression⟩`)`

⟨Fender⟩ and ⟨output expression⟩ must be ⟨expression⟩s.

A ⟨pattern⟩ is an identifier, constant, or one of the following.

`(`⟨pattern⟩ `...)`
`(`⟨pattern⟩ ⟨pattern⟩ `...` `.` ⟨pattern⟩`)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...` `.` ⟨pattern⟩`)`
`#(`⟨pattern⟩ `...)`
`#(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...)`

An ⟨ellipsis⟩ is the identifier "`...`" (three periods).

An identifier appearing within a ⟨pattern⟩ may be an underscore (`_`), a literal identifier listed in the list of literals (⟨literal⟩ `...`), or an ellipsis (`...`). All other identifiers appearing within a ⟨pattern⟩ are *pattern variables*. It is a syntax violation if an ellipsis or underscore appears in (⟨literal⟩ `...`).

Pattern variables match arbitrary input subforms and are used to refer to elements of the input. It is a syntax violation if the same pattern variable appears more than once in a ⟨pattern⟩.

Underscores also match arbitrary input subforms but are not pattern variables and so cannot be used to refer to those elements. Multiple underscores may appear in a ⟨pattern⟩.

A literal identifier matches an input subform if and only if the input subform is an identifier and either both its occurrence in the input expression and its occurrence in the list of literals have the same lexical binding, or the two identifiers have the same name and both have no lexical binding.

A subpattern followed by an ellipsis can match zero or more elements of the input.

More formally, an input form F matches a pattern P if and only if one of the following holds:

- P is an underscore (`_`).
- P is a pattern variable.
- P is a literal identifier and F is an equivalent identifier in the sense of `free-identifier=?` (section 10.5).
- P is of the form `(`P_1 `...` P_n`)` and F is a list of n elements that match P_1 through P_n.
- P is of the form `(`P_1 `...` P_n `.` P_x`)` and F is a list or improper list of n or more elements whose first n elements match P_1 through P_n and whose nth cdr matches P_x.
- P is of the form `(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a proper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.

- P is of the form `(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n `.` P_x`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a list or improper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, whose next $n-m$ elements match P_{m+1} through P_n, and whose nth and final cdr matches P_x.

- P is of the form `#(`P_1 `...` P_n`)` and F is a vector of n elements that match P_1 through P_n.

- P is of the form `#(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a vector of n or more elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.

- P is a pattern datum (any nonlist, nonvector, nonsymbol datum) and F is equal to P in the sense of the `equal?` procedure.

Semantics: `syntax-case` first evaluates ⟨expression⟩. It then attempts to match the ⟨pattern⟩ from the first ⟨clause⟩ against the resulting value, which is unwrapped as necessary to perform the match. If the pattern matches the value and no ⟨fender⟩ is present, ⟨output expression⟩ is evaluated and its value returned as the value of the `syntax-case` expression. If the pattern does not match the value, `syntax-case` tries the second ⟨clause⟩, then the third, and so on. It is a syntax violation if the value does not match any of the patterns.

If the optional ⟨fender⟩ is present, it serves as an additional constraint on acceptance of a clause. If the ⟨pattern⟩ of a given ⟨clause⟩ matches the input value, the corresponding ⟨fender⟩ is evaluated. If ⟨fender⟩ evaluates to a true value, the clause is accepted; otherwise, the clause is rejected as if the pattern had failed to match the value. Fenders are logically a part of the matching process, i.e., they specify additional matching constraints beyond the basic structure of the input.

Pattern variables contained within a clause's ⟨pattern⟩ are bound to the corresponding pieces of the input value within the clause's ⟨fender⟩ (if present) and ⟨output expression⟩. Pattern variables can be referenced only within `syntax` expressions (see below). Pattern variables occupy the same name space as program variables and keywords.

`(syntax` ⟨template⟩`)` syntax

Note: `#'`⟨template⟩ is equivalent to `(syntax` ⟨template⟩`)`.

A `syntax` expression is similar to a `quote` expression except that (1) the values of pattern variables appearing within ⟨template⟩ are inserted into ⟨template⟩, (2) contextual information associated both with the input and with the template is retained in the output to support lexical scoping, and (3) the value of a `syntax` expression is a syntax object.

A ⟨template⟩ is a pattern variable, an identifier that is not a pattern variable, a pattern datum, or one of the following.

`(`⟨subtemplate⟩ `...)`
`(`⟨subtemplate⟩ `...` `.` ⟨template⟩`)`
`#(`⟨subtemplate⟩ `...)`

A ⟨subtemplate⟩ is a ⟨template⟩ followed by zero or more ellipses.

The value of a `syntax` form is a copy of ⟨template⟩ in which the pattern variables appearing within the template are replaced with the input subforms to which they are bound. Pattern data and identifiers that are not pattern variables or ellipses are copied directly into the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the subtemplate than in the associated subpattern, the input form is replicated as necessary. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis, and for at least one such pattern variable, the subtemplate must be followed by exactly as many ellipses as the subpattern in which the pattern variable appears. (Otherwise, the expander would not be able to determine how many times the subform should be repeated in the output.) It is a syntax violation if the constraints of this paragraph are not met.

A template of the form `(`⟨ellipsis⟩ ⟨template⟩`)` is identical to ⟨template⟩, except that ellipses within the template have no special meaning. That is, any ellipses contained within ⟨template⟩ are treated as ordinary identifiers. In particular, the template `(... ...)` produces a single ellipsis. This allows macro uses to expand into forms containing ellipses.

The output produced by `syntax` is wrapped or unwrapped according to the following rules.

- the copy of `(`⟨t_1⟩ `.` ⟨t_2⟩`)` is a pair if ⟨t_1⟩ or ⟨t_2⟩ contain any pattern variables,

- the copy of `(`⟨t⟩ ⟨ellipsis⟩`)` is a list if ⟨t⟩ contains any pattern variables,

- the copy of `#(`⟨t_1⟩ `...` ⟨t_n⟩`)` is a vector if any of ⟨t_1⟩, ..., ⟨t_n⟩ contain any pattern variables, and

- the copy of any portion of ⟨t⟩ not containing any pattern variables is a wrapped syntax object.

The input subforms inserted in place of the pattern variables are wrapped if and only if the corresponding input subforms are wrapped.

The following definitions of `or` illustrate `syntax-case` and `syntax`. The second is equivalent to the first but uses the `#'` prefix instead of the full `syntax` form.

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_) (syntax #f)]
      [(_ e) (syntax e)]
      [(_ e1 e2 e3 ...)
       (syntax (let ([t e1])
                 (if t t (or e2 e3 ...))))])))

(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_) #'#f]
      [(_ e) #'e]
      [(_ e1 e2 e3 ...)
       #'(let ([t e1])
           (if t t (or e2 e3 ...)))])))
```

The examples below define *identifier macros*, macro uses supporting keyword references that do not necessarily appear in the first position of a list-structured form. The second example uses `make-variable-transformer` to handle the case where the keyword appears on the left-hand side of a `set!` expression.

```
(define p (cons 4 5))
(define-syntax p.car
  (lambda (x)
    (syntax-case x ()
      [(_ . rest) #'((car p) . rest)]
      [_  #'(car p)])))
p.car                       ⟹ 4
(set! p.car 15)             ⟹ &syntax exception

(define p (cons 4 5))
(define-syntax p.car
  (make-variable-transformer
    (lambda (x)
      (syntax-case x (set!)
        [(set! _ e) #'(set-car! p e)]
        [(_ . rest) #'((car p) . rest)]
        [_  #'(car p)]))))
(set! p.car 15)
p.car                       ⟹ 15
p                           ⟹ (15 5)
```

10.5. Identifier predicates

`(identifier?` *obj*`)` procedure

Returns `#t` if *obj* is an identifier, i.e., a syntax object representing an identifier, and `#f` otherwise.

The `identifier?` procedure is often used within a fender to verify that certain subforms of an input form are identifiers, as in the definition of `rec`, which creates self-contained recursive objects, below.

```
(define-syntax rec
  (lambda (x)
    (syntax-case x ()
      [(_ x e)
       (identifier? #'x)
       #'(letrec ([x e]) x)])))

(map (rec fact
       (lambda (n)
         (if (= n 0)
             1
             (* n (fact (- n 1))))))
     '(1 2 3 4 5))
          ⟹ (1 2 6 24 120)

(rec 5 (lambda (x) x))      ⟹ &syntax exception
```

The procedures `bound-identifier=?` and `free-identifier=?` each take two identifier arguments and return `#t` if their arguments are equivalent and `#f` otherwise. These predicates are used to compare identifiers according to their *intended use* as free references or bound identifiers in a given context.

`(bound-identifier=?` id_1 id_2`)` procedure

Id_1 and id_2 must be identifiers. The procedure `bound-identifier=?` returns true if and only if a binding for one would capture a reference to the other in the output of the transformer, assuming that the reference appears within the scope of the binding. In general, two identifiers are `bound-identifier=?` only if both are present in the original program or both are introduced by the same transformer application (perhaps implicitly—see `datum->syntax`). Operationally, two identifiers are considered equivalent by `bound-identifier=?` if and only if they have the same name and same marks (section 10.1).

The `bound-identifier=?` procedure can be used for detecting duplicate identifiers in a binding construct or for other preprocessing of a binding construct that requires detecting instances of the bound identifiers.

`(free-identifier=?` id_1 id_2`)` procedure

Id_1 and id_2 must be identifiers. The `free-identifier=?` procedure returns `#t` if and only if the two identifiers would resolve to the same binding if both were to appear in the output of a transformer outside of any bindings inserted by the transformer. (If neither of two like-named identifiers resolves to a binding, i.e., both are unbound,

they are considered to resolve to the same binding.) Operationally, two identifiers are considered equivalent by `free-identifier=?` if and only the topmost matching substitution for each maps to the same binding (section 10.1) or the identifiers have the same name and no matching substitution.

`syntax-case` and `syntax-rules` use `free-identifier=?` to compare identifiers listed in the literals list against input identifiers.

The following definition of unnamed `let` uses `bound-identifier=?` to detect duplicate identifiers.

```
(define-syntax let
  (lambda (x)
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (let notmem?
                       ([x (car ls)] [ls (cdr ls)])
                   (or (null? ls)
                       (and (not (bound-identifier=?
                                  x (car ls)))
                            (notmem? x (cdr ls)))))
                 (unique-ids? (cdr ls))))))
    (syntax-case x ()
      [(_ ((i v) ...) e1 e2 ...)
       (unique-ids? #'(i ...))
       #'((lambda (i ...) e1 e2 ...) v ...)])))
```

The argument `#'(i ...)` to `unique-ids?` is guaranteed to be a list by the rules given in the description of `syntax` above.

With this definition of `let`:

```
(let ([a 3] [a 4]) (+ a a))
          ⟹ &syntax exception
```

However,

```
(let-syntax
  ([dolet (lambda (x)
            (syntax-case x ()
              [(_ b)
               #'(let ([a 3] [b 4]) (+ a b))]))])
  (dolet a))
          ⟹ 7
```

since the identifier `a` introduced by `dolet` and the identifier `a` extracted from the input form are not `bound-identifier=?`.

The following definition of `case` is equivalent to the one in section 10.4. Rather than including `else` in the literals list as before, this version explicitly tests for `else` using `free-identifier=?`.

```
(define-syntax case
  (lambda (x)
    (syntax-case x ()
      [(_ e0 [(k ...) e1 e2 ...] ...
             [else-key else-e1 else-e2 ...])
       (and (identifier? #'else-key)
            (free-identifier=? #'else-key #'else))
       #'(let ([t e0])
           (cond
             [(memv t '(k ...)) e1 e2 ...]
             ...
             [else else-e1 else-e2 ...]))]
      [(_ e0 [(ka ...) e1a e2a ...]
             [(kb ...) e1b e2b ...] ...)
       #'(let ([t e0])
           (cond
             [(memv t '(ka ...)) e1a e2a ...]
             [(memv t '(kb ...)) e1b e2b ...]
             ...))])))
```

With either definition of `case`, `else` is not recognized as an auxiliary keyword if an enclosing lexical binding for `else` exists. For example,

```
(let ([else #f])
  (case 0 [else (write "oops")]))
          ⟹ &syntax exception
```

since `else` is bound lexically and is therefore not the same `else` that appears in the definition of `case`.

10.6. Syntax-object and datum conversions

`(syntax->datum` *syntax-object*`)` procedure

The procedure `syntax->datum` strips all syntactic information from a syntax object and returns the corresponding Scheme datum.

Identifiers stripped in this manner are converted to their symbolic names, which can then be compared with `eq?`. Thus, a predicate `symbolic-identifier=?` might be defined as follows.

```
(define symbolic-identifier=?
  (lambda (x y)
    (eq? (syntax->datum x)
         (syntax->datum y))))
```

`(datum->syntax` *template-id datum*`)` procedure

Template-id must be a template identifier and *datum* should be a datum value. The `datum->syntax` procedure returns a syntax object representation of *datum* that contains the same contextual information as *template-id*, with the effect that the syntax object behaves as if it were introduced into the code when *template-id* was introduced.

The `datum->syntax` procedure allows a transformer to "bend" lexical scoping rules by creating *implicit identifiers* that behave as if they were present in the input form,

thus permitting the definition of macros that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form. For example, the following defines a `loop` expression that uses this controlled form of identifier capture to bind the variable `break` to an escape procedure within the loop body. (The derived `with-syntax` form is like `let` but binds pattern variables—see section 10.8.)

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(k e ...)
       (with-syntax
           ([break (datum->syntax #'k 'break)])
         #'(call-with-current-continuation
             (lambda (break)
               (let f () e ... (f)))))])))

(let ((n 3) (ls '()))
  (loop
    (if (= n 0) (break ls))
    (set! ls (cons 'a ls))
    (set! n (- n 1))))
      ⟹ (a a a)
```

Were `loop` to be defined as

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(_ e ...)
       #'(call-with-current-continuation
           (lambda (break)
             (let f () e ... (f))))])))
```

the variable `break` would not be visible in `e ...`.

The datum argument *datum* may also represent an arbitrary Scheme form, as demonstrated by the following definition of `include`.

```
(define-syntax include
  (lambda (x)
    (define read-file
      (lambda (fn k)
        (let ([p (open-file-input-port fn)])
          (let f ([x (get-datum p)])
            (if (eof-object? x)
                (begin (close-port p) '())
                (cons (datum->syntax k x)
                      (f (get-datum p))))))))
    (syntax-case x ()
      [(k filename)
       (let ([fn (syntax->datum #'filename)])
         (with-syntax ([(exp ...)
                        (read-file fn #'k)])
           #'(begin exp ...)))])))
```

`(include "filename")` expands into a `begin` expression containing the forms found in the file named by `"filename"`. For example, if the file `flib.ss` contains `(define f (lambda (x) (g (* x x))))`, and the file `glib.ss` contains `(define g (lambda (x) (+ x x)))`, the expression

```
(let ()
  (include "flib.ss")
  (include "glib.ss")
  (f 5))
```

evaluates to `50`.

The definition of `include` uses `datum->syntax` to convert the objects read from the file into syntax objects in the proper lexical context, so that identifier references and definitions within those expressions are scoped where the `include` form appears.

Using `datum->syntax`, it is even possible to break hygiene entirely and write macros in the style of old Lisp macros. The `lisp-transformer` procedure defined below creates a transformer that converts its input into a datum, calls the programmer's procedure on this datum, and converts the result back into a syntax object that is scoped at top level (or, more accurately, wherever `lisp-transformer` is defined).

```
(define lisp-transformer
  (lambda (p)
    (lambda (x)
      (datum->syntax #'lisp-transformer
        (p (syntax->datum x))))))
```

10.7. Generating lists of temporaries

Transformers can introduce a fixed number of identifiers into their output simply by naming each identifier. In some cases, however, the number of identifiers to be introduced depends upon some characteristic of the input expression. A straightforward definition of `letrec`, for example, requires as many temporary identifiers as there are binding pairs in the input expression. The procedure `generate-temporaries` is used to construct lists of temporary identifiers.

`(generate-temporaries` *l*`)` procedure

L must be be a list or syntax object representing a list-structured form; its contents are not important. The number of temporaries generated is the number of elements in *l*. Each temporary is guaranteed to be unique, i.e., different from all other identifiers.

A definition of `letrec` that uses `generate-temporaries` is shown below.

```
(define-syntax letrec
  (lambda (x)
    (syntax-case x ()
```

```
((_ ((i v) ...) e1 e2 ...)
 (with-syntax
      (((t ...)
        (generate-temporaries (syntax (i ...)))))
    (syntax (let ((i #f) ...)
              (let ((t v) ...)
                (set! i t) ...
                (let () e1 e2 ...)))))))))
```

Any transformer that uses `generate-temporaries` in this fashion can be rewritten to avoid using it, albeit with a loss of clarity. The trick is to use a recursively defined intermediate form that generates one temporary per expansion step and completes the expansion after enough temporaries have been generated. See the definition for `letrec` in report appendix A.

10.8. Derived forms and procedures

The forms and procedures described in this section are *derived*, i.e., they can defined in terms of the forms and procedures described in earlier sections of this document.

`(with-syntax ((`⟨pattern⟩ ⟨expression⟩`) ...)` ⟨body⟩`)` syntax

The derived `with-syntax` form is used to bind pattern variables, just as `let` is used to bind variables. This allows a transformer to construct its output in separate pieces, then put the pieces together.

Each ⟨pattern⟩ is identical in form to a `syntax-case` pattern. The value of each ⟨expression⟩ is computed and destructured according to the corresponding ⟨pattern⟩, and pattern variables within the ⟨pattern⟩ are bound as with `syntax-case` to the corresponding portions of the value within ⟨body⟩.

The `with-syntax` form may be defined in terms of `syntax-case` as follows.

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((_ ((p e0) ...) e1 e2 ...)
       (syntax (syntax-case (list e0 ...) ()
                 ((p ...) (let () e1 e2 ...))))))))
```

The following definition of `cond` demonstrates the use of `with-syntax` to support transformers that employ recursion internally to construct their output. It handles all `cond` clause variations and takes care to produce one-armed `if` expressions where appropriate.

```
(define-syntax cond
  (lambda (x)
    (syntax-case x ()
      [(_ c1 c2 ...)
       (let f ([c1 #'c1] [c2* #'(c2 ...)])
         (syntax-case c2* ()
           [()
            (syntax-case c1 (else =>)
              [(else e1 e2 ...) #'(begin e1 e2 ...)]
              [(e0) #'(let ([t e0]) (if t t))]
              [(e0 => e1)
               #'(let ([t e0]) (if t (e1 t)))]
              [(e0 e1 e2 ...)
               #'(if e0 (begin e1 e2 ...))])]
           [(c2 c3 ...)
            (with-syntax ([rest (f #'c2 #'(c3 ...))])
              (syntax-case c1 (=>)
                [(e0) #'(let ([t e0]) (if t t rest))]
                [(e0 => e1)
                 #'(let ([t e0]) (if t (e1 t) rest))]
                [(e0 e1 e2 ...)
                 #'(if e0
                       (begin e1 e2 ...)
                       rest)]))])))])))
```

`(quasisyntax` ⟨template⟩`)` syntax

Note: `#`⟨template⟩ is equivalent to `(quasisyntax` ⟨template⟩`)`, `#,`⟨template⟩ is equivalent to `(unsyntax` ⟨template⟩`)`, and `#,@`⟨template⟩ is equivalent to `(unsyntax-splicing` ⟨template⟩`)`.

The `quasisyntax` form is similar to `syntax`, but it allows parts of the quoted text to be evaluated, in a manner similar to the operation of `quasiquote` (report section 9.19).

Within a `quasisyntax` *template*, subforms of `unsyntax` and `unsyntax-splicing` forms are evaluated, and everything else is treated as ordinary template material, as with `syntax`. The value of each `unsyntax` subform is inserted into the output in place of the `unsyntax` form, while the value of each `unsyntax-splicing` subform is spliced into the surrounding list or vector structure. Uses of `unsyntax` and `unsyntax-splicing` are valid only within `quasisyntax` expressions.

A `quasisyntax` expression may be nested, with each `quasisyntax` introducing a new level of syntax quotation and each `unsyntax` or `unsyntax-splicing` taking away a level of quotation. An expression nested within n `quasisyntax` expressions must be within n `unsyntax` or `unsyntax-splicing` expressions to be evaluated.

The `quasisyntax` keyword can be used in place of `with-syntax` in many cases. For example, the definition of `case` shown under the description of `with-syntax` above can be rewritten using `quasisyntax` as follows.

```
(define-syntax case
  (lambda (x)
    (syntax-case x ()
      [(_ e c1 c2 ...)
       #`(let ([t e])
```

```
#,(let f ([c1 #'c1] [cmore #'(c2 ...)])
    (if (null? cmore)
        (syntax-case c1 (else)
          [(else e1 e2 ...)
           #'(begin e1 e2 ...)]
          [((k ...) e1 e2 ...)
           #'(if (memv t '(k ...))
                 (begin e1 e2 ...))])
        (syntax-case c1 ()
          [((k ...) e1 e2 ...)
           #`(if (memv t '(k ...))
                 (begin e1 e2 ...)
                 #,(f (car cmore)
                      (cdr cmore)))]))))))))
```

Uses of `unsyntax` and `unsyntax-splicing` with zero or more than one subform are valid only in splicing (list or vector) contexts. (`unsyntax` *template* `...`) is equivalent to (`unsyntax` *template*) `...`, and (`unsyntax-splicing` *template* `...`) is equivalent to (`unsyntax-splicing` *template*) `...`. These forms are primarily useful as intermediate forms in the output of the `quasisyntax` expander.

Note: Uses of `unsyntax` and `unsyntax-splicing` with zero or more than one subform enable certain idioms [2], such as `#,@#,@`, which has the effect of a doubly indirect splicing when used within a doubly nested and doubly evaluated `quasisyntax` expression, as with the nested `quasiquote` examples shown in section 9.19.

Note: Any `syntax-rules` form can be expressed with `syntax-case` by making the `lambda` expression and `syntax` expressions explicit, and `syntax-rules` may be defined in terms of `syntax-case` as follows.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      [(_ (k ...) [(_ . p) f ... t] ...)
       #'(lambda (x)
           (syntax-case x (k ...)
             [(_ . p) f ... #'t] ...))])))
```

A more robust implementation would verify that the literals ⟨literal⟩ `...` are all identifiers, that the first position of each pattern is an identifier, and that at most one fender is present in each clause.

Note: The `identifier-syntax` form of the base library (see report section 9.21) may be defined in terms of `syntax-case`, `syntax`, and `make-variable-transformer` as follows.

```
(define-syntax identifier-syntax
  (syntax-rules (set!)
    [(_ e)
     (lambda (x)
       (syntax-case x ()
         [id (identifier? #'id) #'e]
         [(_ x (... ...)) #'(e x (... ...))]))]
    [(_ (id exp1) ((set! var val) exp2))
     (and (identifier? #'id) (identifier? #'var))
     (make-variable-transformer
       (lambda (x)
         (syntax-case x (set!)
           [(set! var val) #'exp2]
           [(id x (... ...)) #'(exp1 x (... ...))]
           [id (identifier? #'id) #'exp1])))]))
```

10.9. Syntax violations

(`syntax-violation` *who message form*) procedure
(`syntax-violation` *who message form subform*) procedure

Who must be `#f` or a string or a symbol. *Message* must be a string. *Form* must be a syntax object or a datum value. *Subform* must be a syntax object or a datum value. The `syntax-violation` procedure raises an exception, reporting a syntax violation. The *who* argument should describe the macro transformer that detected the exception. The *message* argument should describe the violation. The *form* argument is the erroneous source syntax object or a datum value representing a form. The optional *subform* argument is a syntax object or datum value representing a form that more precisely locates the violation.

If *who* is `#f`, `syntax-violation` attempts to infer an appropriate value for the condition object (see below) as follows: When *form* is either an identifier or a list-structured syntax object containing an identifier as its first element, then the inferred value is the identifier's symbol. Otherwise, no value for *who* is provided as part of the condition object.

The condition object provided with the exception (see chapter 6) has the following condition types:

- If *who* is not `#f` or can be inferred, the condition has condition type `&who`, with *who* as the value of the `who` field. In that case, *who* should identify the procedure or entity that detected the exception. If it is `#f`, the condition does not have condition type `&who`.
- The condition has condition type `&message`, with *message* as the value of the `message` field.
- The condition has condition type `&syntax` with *form* as the value of the `form` field, and *subform* as the value of the `subform` field. If *subform* is not provided, the value of the `subform` field is `#f`.

11. Hash tables

The (`r6rs hash-tables`) library provides hash tables. A *hash table* is a data structure that associates keys with values. Any object can be used as a key, provided a *hash function* and a suitable *equivalence function* is available. A hash

function is a procedure that maps keys to integers, and must be compatible with the equivalence function, which is a procedure that accepts two keys and returns true if they are equivalent, otherwise returns `#f`. Standard hash tables for arbitrary objects based on the `eq?` and `eqv?` predicates (see report section 9.6) are provided. Also, standard hash functions for several types are provided.

This section uses the *hash-table* parameter name for arguments that must be hash tables, and the *key* parameter name for arguments that must be hash-table keys.

11.1. Constructors

`(make-eq-hash-table)` procedure
`(make-eq-hash-table` *k*`)` procedure

Returns a newly allocated mutable hash table that accepts arbitrary objects as keys, and compares those keys with `eq?`. If an argument is given, the initial capacity of the hash table is set to approximately *k* elements.

`(make-eqv-hash-table)` procedure
`(make-eqv-hash-table` *k*`)` procedure

Returns a newly allocated mutable hash table that accepts arbitrary objects as keys, and compares those keys with `eqv?`. If an argument is given, the initial capacity of the hash table is set to approximately *k* elements.

`(make-hash-table` *hash-function equiv*`)` procedure
`(make-hash-table` *hash-function equiv k*`)` procedure

Hash-function and *equiv* must be procedures. *Hash-function* will be called by other procedures described in this chapter with a key as argument, and must return a non-negative exact integer. *Equiv* will be called by other procedures described in this chapter with two keys as arguments. The `make-hash-table` procedure returns a newly allocated mutable hash table using *hash-function* as the hash function and *equiv* as the equivalence function used to compare keys. If a third argument is given, the initial capacity of the hash table is set to approximately *k* elements.

Both the hash function *hash-function* and the equivalence function *equiv* should behave like pure functions on the domain of keys. For example, the `string-hash` and `string=?` procedures are permissible only if all keys are strings and the contents of those strings are never changed so long as any of them continues to serve as a key in the hash table. Furthermore, any pair of values for which the equivalence function *equiv* returns true should be hashed to the same exact integers by *hash-function*.

Note: Hash tables are allowed to cache the results of calling the hash function and equivalence function, so programs cannot rely on the hash function being called for every lookup or update. Furthermore any hash-table operation may call the hash function more than once.

Rationale: Hash-table lookups are often followed by updates, so caching may improve performance. Hash tables are free to change their internal representation at any time, which may result in many calls to the hash function.

11.2. Procedures

`(hash-table?` *hash-table*`)` procedure

Returns `#t` if *hash-table* is a hash table. Otherwise, returns `#f`.

`(hash-table-size` *hash-table*`)` procedure

Returns the number of keys contained in *hash-table* as an exact integer.

`(hash-table-ref` *hash-table key default*`)` procedure

Returns the value in *hash-table* associated with *key*. If *hash-table* does not contain an association for *key*, then *default* is returned.

`(hash-table-set!` *hash-table key obj*`)` procedure

Changes *hash-table* to associate *key* with *obj*, adding a new association or replacing any existing association for *key*, and returns the unspecified value.

`(hash-table-delete!` *hash-table key*`)` procedure

Removes any association for *key* within *hash-table*, and returns the unspecified value.

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

Returns `#t` if *hash-table* contains an association for *key*. Otherwise, returns `#f`.

`(hash-table-update!` *hash-table key proc default*`)` procedure

Proc must be a procedure that takes a single argument. The `hash-table-update!` procedure applies *proc* to the value in *hash-table* associated with *key*, or to *default* if *hash-table* does not contain an association for *key*. The *hash-table* is then changed to associate *key* with the result of *proc*.

The behavior of `hash-table-update!` is equivalent to the following code, but may be implemented more efficiently in cases where the implementation can avoid multiple lookups of the same key:

```
(hash-table-set!
 hash-table key
 (proc (hash-table-ref
        hash-table key default)))
```

`(hash-table-copy` *hash-table*`)` procedure
`(hash-table-copy` *hash-table immutable*`)` procedure

Returns a copy of *hash-table*. If the *immutable* argument is provided and is true, the returned hash table is immutable; otherwise it is mutable.

Rationale: Hash table references may be less expensive with immutable hash tables. Also, the creator of a hash table may wish to prevent modifications, particularly by code outside of the creator's control.

`(hash-table-clear!` *hash-table*`)` procedure
`(hash-table-clear!` *hash-table k*`)` procedure

Removes all associations from *hash-table* and returns the unspecified value.

If a second argument is given, the current capacity of the hash table is reset to approximately k elements.

`(hash-table-keys` *hash-table*`)` procedure

Returns a vector of all keys in *hash-table*. The order of the vector is unspecified.

`(hash-table-entries` *hash-table*`)` procedure

Returns two values, a vector of the keys in *hash-table*, and a vector of the corresponding values.

```
(let ((h (make-eqv-hash-table)))
  (hash-table-set! h 1 'one)
  (hash-table-set! h 2 'two)
  (hash-table-set! h 3 'three)
  (hash-table-entries h))
        ⟹ #(1 2 3), #(one two three)
        ; two return values
```

11.3. Inspection

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

Returns the equivalence function used by *hash-table* to compare keys. For hash tables created with `make-eq-hash-table` and `make-eqv-hash-table`, returns `eq?` and `eqv?` respectively.

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

Returns the hash function used by *hash-table*. For hash tables created by `make-eq-hash-table` or `make-eqv-hash-table`, `#f` is returned.

Rationale: The `make-eq-hash-table` and `make-eqv-hash-table` constructors are designed to hide their hash function. This allows implementations to use the machine address of an object as its hash value, rehashing parts of the table as necessary whenever the garbage collector moves objects to a different address.

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

Returns `#t` if *hash-table* is mutable, otherwise returns `#f`.

11.4. Hash functions

The `equal-hash`, `string-hash`, and `string-ci-hash` procedures of this section are acceptable as hash functions only if the keys on which they are called do not suffer side effects while the hash table remains in use.

`(equal-hash` *obj*`)` procedure

Returns an integer hash value for *obj*, based on its structure and current contents. This hash function is suitable for use with `equal?` as an equivalence function.

`(string-hash` *string*`)` procedure

Returns an integer hash value for *string*, based on its current contents. This hash function is suitable for use with `string=?` as an equivalence function.

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

Returns an integer hash value for *string* based on its current contents, ignoring case. This hash function is suitable for use with `string-ci=?` as an equivalence function.

`(symbol-hash` *symbol*`)` procedure

Returns an integer hash value for *symbol*.

12. Enumerations

This chapter describes the `(r6rs enum)` library for dealing with enumerated values and sets of enumerated values. Enumerated values are represented by ordinary symbols, while finite sets of enumerated values form a separate type, known as the *enumeration sets*. The enumeration sets are further partitioned into sets that share the same *universe* and *enumeration type*. These universes and enumeration types are created by the `make-enumeration`

procedure. Each call to that procedure creates a new enumeration type.

This library interprets each enumeration set with respect to its specific universe of symbols and enumeration type. This facilitates efficient implementation of enumeration sets and enables the complement operation.

In the definition of the following procedures, let *enum-set* range over the enumeration sets, which are defined as the subsets of the universes that can be defined using `make-enumeration`.

`(make-enumeration` *list*`)` procedure

List must be a list of symbols. The `make-enumeration` procedure creates a new enumeration type whose universe consists of those symbols (in canonical order of their first appearance in the list) and returns that universe as an enumeration set whose universe is itself and whose enumeration type is the newly created enumeration type.

`(enum-set-universe` *enum-set*`)` procedure

Returns the set of all symbols that comprise the universe of its argument.

`(enum-set-indexer` *enum-set*`)` procedure

Returns a unary procedure that, given a symbol that is in the universe of *enum-set*, returns its 0-origin index within the canonical ordering of the symbols in the universe; given a value not in the universe, the unary procedure returns `#f`.

```
(let* ((e (make-enumeration '(red green blue)))
       (i (enum-set-indexer e)))
  (list (i 'red) (i 'green) (i 'blue) (i 'yellow)))
        ⟹ (0 1 2 #f)
```

The `enum-set-indexer` procedure could be defined as follows (using the `memq` procedure from the `(r6rs lists)` library):

```
(define (enum-set-indexer set)
  (let* ((symbols (enum-set->list
                   (enum-set-universe set)))
         (cardinality (length symbols)))
    (lambda (x)
      (let ((probe (memq x symbols)))
        (if probe
            (- cardinality (length probe))
            #f)))))
```

`(enum-set-constructor` *enum-set*`)` procedure

Returns a unary procedure that, given a list of symbols that belong to the universe of *enum-set*, returns a subset of that universe that contains exactly the symbols in the list. If any value in the list is not a symbol that belongs to the universe, then the unary procedure raises an exception with condition type `&assertion`.

`(enum-set->list` *enum-set*`)` procedure

Returns a list of the symbols that belong to its argument, in the canonical order of the universe of *enum-set*.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (enum-set->list (c '(blue red))))
        ⟹ (red blue)
```

`(enum-set-member?` *symbol enum-set*`)` procedure
`(enum-set-subset?` *enum-set*$_1$ *enum-set*$_2$`)` procedure
`(enum-set=?` *enum-set*$_1$ *enum-set*$_2$`)` procedure

The `enum-set-member?` procedure returns `#t` if its first argument is an element of its second argument, `#f` otherwise.

The `enum-set-subset?` procedure returns `#t` if the universe of *enum-set*$_1$ is a subset of the universe of *enum-set*$_2$ (considered as sets of symbols) and every element of *enum-set*$_1$ is a member of *enum-set*$_2$. It returns `#f` otherwise.

The `enum-set=?` procedure returns `#t` if *enum-set*$_1$ is a subset of *enum-set*$_2$ and vice versa, as determined by the `enum-set-subset?` procedure. This implies that the universes of the two sets are equal as sets of symbols, but does not imply that they are equal as enumeration types. Otherwise, `#f` is returned.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (list
   (enum-set-member? 'blue (c '(red blue)))
   (enum-set-member? 'green (c '(red blue)))
   (enum-set-subset? (c '(red blue)) e)
   (enum-set-subset? (c '(red blue)) (c '(blue red)))
   (enum-set-subset? (c '(red blue)) (c '(red)))
   (enum-set=? (c '(red blue)) (c '(blue red)))))
            ⟹ (#t #f #t #t #f #t)
```

`(enum-set-union` *enum-set*$_1$ *enum-set*$_2$`)` procedure
`(enum-set-intersection` *enum-set*$_1$ *enum-set*$_2$`)` procedure
`(enum-set-difference` *enum-set*$_1$ *enum-set*$_2$`)` procedure

Enum-set$_1$ and *enum-set*$_2$ must be enumeration sets that have the same enumeration type. If their enumeration types differ, a `&assertion` violation is raised.

The `enum-set-union` procedure returns the union of *enum-set*$_1$ and *enum-set*$_2$. The `enum-set-intersection`

procedure returns the intersection of *enum-set*$_1$ and *enum-set*$_2$. The `enum-set-difference` procedure returns the difference of *enum-set*$_1$ and *enum-set*$_2$.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (list (enum-set->list
         (enum-set-union (c '(blue)) (c '(red))))
        (enum-set->list
         (enum-set-intersection (c '(red green))
                                (c '(red blue))))
        (enum-set->list
         (enum-set-difference (c '(red green))
                              (c '(red blue))))))

          ⟹ ((red blue) (red) (green))
```

`(enum-set-complement` *enum-set*`)` procedure

Returns *enum-set*'s complement with respect to its universe.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (enum-set->list
   (enum-set-complement (c '(red)))))
                          ⟹ (green blue)
```

`(enum-set-projection` *enum-set*$_1$ *enum-set*$_2$`)` procedure

Projects *enum-set*$_1$ into the universe of *enum-set*$_2$, dropping any elements of *enum-set*$_1$ that do not belong to the universe of *enum-set*$_2$. (If *enum-set*$_1$ is a subset of the universe of its second, then no elements are dropped, and the injection is returned.)

```
(let ((e1 (make-enumeration
           '(red green blue black)))
      (e2 (make-enumeration
           '(red black white))))
  (enum-set->list
   (enum-set-projection e1 e2))))
                          ⟹ (red black)
```

`(define-enumeration` ⟨type-name⟩ syntax
`(`⟨symbol⟩ ...`)`
⟨constructor-syntax⟩`)`

The `define-enumeration` form defines an enumeration type and provides two macros for constructing its members and sets of its members.

A `define-enumeration` form is a definition and can appear anywhere any other ⟨definition⟩ can appear.

⟨Type-name⟩ is an identifier that is bound as a syntactic keyword; ⟨symbol⟩ ... are the symbols that comprise the universe of the enumeration (in order).

`(`⟨type-name⟩ ⟨symbol⟩`)` checks at macro-expansion time whether ⟨symbol⟩ is in the universe associated with ⟨type-name⟩. If it is, then `(`⟨type-name⟩ ⟨symbol⟩`)` is equivalent to ⟨symbol⟩. It is a syntax violation if it is not.

⟨Constructor-syntax⟩ is an identifier that is bound to a macro that, given any finite sequence of the symbols in the universe, possibly with duplicates, expands into an expression that evaluates to the enumeration set of those symbols.

`(`⟨constructor-syntax⟩ ⟨symbol⟩ ...`)` checks at macro-expansion time whether every ⟨symbol⟩ ... is in the universe associated with ⟨type-name⟩. It is a syntax violation if one or more is not. Otherwise

`(`⟨constructor-syntax⟩ ⟨symbol⟩ ...`)`

is equivalent to

```
((enum-set-constructor (⟨constructor-syntax⟩))
 (list '⟨symbol⟩ ...)).
```

Example:

```
(define-enumeration color
  (black white purple maroon)
  color-set)

(color black)                ⟹ black
(color purpel)               ⟹ &syntax exception
(enum-set->list (color-set))⟹ ()
(enum-set->list
 (color-set maroon white))   ⟹ (white maroon)
```

13. Miscellaneous libraries

13.1. `when` and `unless`

This section describes the `(r6rs when-unless)` library.

`(when` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax
`(unless` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax

Syntax: ⟨Test⟩ must be an expression. *Semantics:* A `when` expression is evaluated by evaluating the ⟨test⟩ expression. If ⟨test⟩ evaluates to a true value, the remaining ⟨expression⟩s are evaluated in order, and the result(s) of the last ⟨expression⟩ is(are) returned as the result(s) of the entire `when` expression. Otherwise, the `when` expression evaluates to the unspecified value. An `unless` expression is evaluated by evaluating the ⟨test⟩ expression. If ⟨test⟩ evaluates to false, the remaining ⟨expression⟩s are evaluated in order, and the result(s) of the last ⟨expression⟩ is(are) returned as the result(s) of the entire `unless` expression. Otherwise, the `unless` expression evaluates to the unspecified value.

```
(when (> 3 2) 'greater)      ⟹ greater
(when (< 3 2) 'greater)      ⟹ the unspecified value
(unless (> 3 2) 'less)       ⟹ the unspecified value
(unless (< 3 2) 'less)       ⟹ less
```

The `when` and `unless` expressions are derived forms. They could be defined in terms of base library forms by the following macros:

```
(define-syntax when
  (syntax-rules ()
    ((when test result1 result2 ...)
     (if test
         (begin result1 result2 ...)))))

(define-syntax unless
  (syntax-rules ()
    ((unless test result1 result2 ...)
     (if (not test)
         (begin result1 result2 ...)))))
```

13.2. case-lambda

This section describes the `(r6rs case-lambda)` library.

`(case-lambda` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntax

Syntax: Each ⟨clause⟩ should be of the form

`(`⟨formals⟩ ⟨body⟩`)`

⟨Formals⟩ must be as in a `lambda` form (report section 9.5.2), ⟨body⟩ must be a body according to report section 9.4.

Semantics: A `case-lambda` expression evaluates to a procedure. This procedure, when applied, tries to match its arguments to the ⟨clause⟩s in order. The arguments match a clause if one the following conditions is fulfilled:

- ⟨Formals⟩ has the form `(`⟨variable⟩ ...`)` and the number of arguments is the same as the number of formal parameters in ⟨formals⟩.
- ⟨Formals⟩ has the form `(`⟨variable$_1$⟩ ... ⟨variable$_n$⟩ `.` ⟨variable$_{n+1}$⟩`)` and the number of arguments is at least n.
- ⟨Formals⟩ has the form ⟨variable⟩.

For the first clause matched by the arguments, the variables of the ⟨formals⟩ are bound to fresh locations containing the argument values in the same arrangement as with `lambda`.

If the arguments match none of the clauses, an exception with condition type `&assertion` is raised.

```
(define foo
  (case-lambda
   (() 'zero)
   ((x) (list 'one x))
   ((x y) (list 'two x y))
   ((a b c d . e) (list 'four a b c d e))
   (rest (list 'rest rest))))

(foo)                        ⟹ zero
(foo 1)                      ⟹ (one 1)
(foo 1 2)                    ⟹ (two 1 2)
(foo 1 2 3)                  ⟹ (rest (1 2 3))
(foo 1 2 3 4)                ⟹ (four 1 2 3 4 ())
```

A sample definition of `case-lambda` in terms of simpler forms is in appendix 18.

13.3. Command-line access and exit values

The procedures described in this section are exported by the `(r6rs programs)` library.

`(command-line)` procedure

Returns a list of strings with at least one element. The first element is an implementation-specific name for the running top-level program. The following elements are command-line arguments according to the operating system's conventions.

`(exit)` procedure
`(exit` *obj*`)` procedure

Exits the running program and communicates an exit value to the operating system. If no argument is supplied, the `exit` procedure should communicate to the operating system that the program exited normally. If an argument is supplied, the `exit` procedure should translate the argument into an appropriate exit value for the operating system.

14. Composite library

The `(r6rs)` library is a composite of most of the libraries described in this report. The only exceptions are:

- `(r6rs records explicit)` (section 5.2)
- `(r6rs mutable-pairs)` (chapter 16)
- `(r6rs eval)` (chapter 15)
- `(r6rs r5rs)` (chapter 17)

The library exports all procedures and syntactic forms provided by the component libraries.

Note: Even though `(r6rs records explicit)` is not included, `(r6rs records implicit)` is included, which subsumes the functionality of `(r6rs records explicit)`.

All of the bindings exported by `(r6rs)` are exported for both `run` and `expand`; see report section 6.2.

15. `eval`

The `(r6rs eval)` library allows a program to create Scheme expressions as data at run time and evaluate them.

`(eval` *expression environment-specifier*`)` procedure

Evaluates *expression* in the specified environment and returns its value. *Expression* must be a valid Scheme expression represented as a datum value, and *environment-specifier* must be a *library specifier*, which can be created using the `environment` procedure described below.

If the first argument to `eval` is not a syntactically correct expression, then `eval` must raise an exception with condition type `&syntax`. Specifically, if the first argument to `eval` is a definition or a splicing `begin` form containing a definition, it must raise an exception with condition type `&syntax`.

`(environment` *import-spec ...*`)` procedure

Import-spec must be a datum representing an ⟨import spec⟩ (see report section 6.1). The `environment` procedure returns an environment corresponding to *import-spec*

The bindings of the environment represented by the specifier are immutable: If `eval` is applied to an expression that attempts to assign to one of the variables of the environment, `eval` must raise an exception with a condition type `&assertion`.

```
(library (foo)
  (export)
  (import (r6rs))
  (write
    (eval '(let ((x 3)) x)
          (environment '(r6rs)))))
            writes 3

(library (foo)
  (export)
  (import (r6rs))
  (write
    (eval
      '(eval:car (eval:cons 2 4))
      (environment
        '(prefix (only (r6rs) car cdr cons null?)
                 eval:)))))
            writes 2
```

16. Mutable pairs

The procedures provided by the `(r6rs mutable-pairs)` library allow new values to be assigned to the car and cdr fields of previously allocated pairs.

16.1. Procedures

`(set-car!` *pair obj*`)` procedure

Stores *obj* in the car field of *pair*. Returns the unspecified value.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)            ⟹  the unspecified value
(set-car! (g) 3)            ⟹  unspecified
          ; should raise &assertion exception
```

If an immutable pair is passed to `set-car!`, an exception with condition type `&assertion` should be raised.

`(set-cdr!` *pair obj*`)` procedure

Stores *obj* in the cdr field of *pair*. Returns the unspecified value.

If an immutable pair is passed to `set-cdr!`, an exception with condition type `&assertion` should be raised.

```
(let ((x (list 'a 'b 'c 'a))
      (y (list 'a 'b 'c 'a 'b 'c 'a)))
  (set-cdr! (list-tail x 2) x)
  (set-cdr! (list-tail y 5) y)
  (list
   (equal? x x)
   (equal? x y)
   (equal? (list x y 'a) (list y x 'b))))
        ⟹  (#t #t #f)
```

17. R^{5}RS compatibility

The procedures described in this chapter are exported from the `(r6rs r5rs)` library, and provide procedures described in the previous revision of this report [5], but omitted from this revision.

`(exact->inexact` *z*`)` procedure
`(inexact->exact` *z*`)` procedure

These are the same as the `->inexact` and `->exact` procedures; see report section 9.9.4.

`(quotient` n_1 n_2`)` procedure
`(remainder` n_1 n_2`)` procedure
`(modulo` n_1 n_2`)` procedure

These procedures implement number-theoretic (integer) division. n_2 must be non-zero. All three procedures return integers. If n_1/n_2 is an integer:

```
(quotient n1 n2)      ⟹ n1/n2
(remainder n1 n2)     ⟹ 0
(modulo n1 n2)        ⟹ 0
```

If n_1/n_2 is not an integer:

```
(quotient n1 n2)      ⟹ nq
(remainder n1 n2)     ⟹ nr
(modulo n1 n2)        ⟹ nm
```

where n_q is n_1/n_2 rounded towards zero, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r and n_m differ from n_1 by a multiple of n_2, n_r has the same sign as n_1, and n_m has the same sign as n_2.

Consequently, for integers n_1 and n_2 with n_2 not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                      ⟹ #t
```

provided all numbers involved in that computation are exact.

```
(modulo 13 4)             ⟹ 1
(remainder 13 4)          ⟹ 1

(modulo -13 4)            ⟹ 3
(remainder -13 4)         ⟹ -1

(modulo 13 -4)            ⟹ -3
(remainder 13 -4)         ⟹ 1

(modulo -13 -4)           ⟹ -1
(remainder -13 -4)        ⟹ -1

(remainder -13 -4.0)      ⟹ -1.0  ; inexact
```

Note: These procedures could be defined in terms of `div` and `mod` (see report section 9.9.4) as follows (without checking of the argument types):

```
(define (sign n)
  (cond
    ((negative? n) -1)
    ((positive? n) 1)
    (else 0)))

(define (quotient n1 n2)
  (* (sign n1) (sign n2) (div (abs n1) (abs n2))))

(define (remainder n1 n2)
  (* (sign n1) (mod (abs n1) (abs n2))))

(define (modulo n1 n2)
  (* (sign n2) (mod (* (sign n2) n1) (abs n2))))
```

`(delay` ⟨expression⟩`)` syntax

The `delay` construct is used together with the procedure `force` to implement *lazy evaluation* or *call by need*. `(delay` ⟨expression⟩`)` returns an object called a *promise* which at some point in the future may be asked (by the `force` procedure) to evaluate ⟨expression⟩, and deliver the resulting value. The effect of ⟨expression⟩ returning multiple values is unspecified.

`(force` *promise*`)` procedure

Promise must be a promise.

Forces the value of *promise*. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned.

```
(force (delay (+ 1 2)))     ⟹ 3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                            ⟹ (3 3)

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail a-stream)))
                            ⟹ 2
```

Promises are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                           ⟹ a promise
(force p)                   ⟹ 6
p                           ⟹ a promise, still
(begin (set! x 10)
       (force p))           ⟹ 6
```

Here is a possible implementation of `delay` and `force`. Promises are implemented here as procedures of no arguments, and `force` simply calls its argument:

```
(define force
  (lambda (object)
    (object)))
```

The expression

```
(delay ⟨expression⟩)
```

has the same meaning as the procedure call

```
(make-promise (lambda () ⟨expression⟩))
```

as follows

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (make-promise (lambda () expression)))))
```

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result))))))))
```

Rationale: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of `make-promise`.

`(null-environment` n`)` procedure

N must be the exact integer 5. The `null-environment` procedure returns an environment specifier suitable for use with `eval` (see chapter 15) representing an environment that is empty except for the (syntactic) bindings for all syntactic keywords described in the previous revision of this report [5].

`(scheme-report-environment` n`)` procedure

N must be the exact integer 5. The `scheme-report-environment` procedure returns an environment specifier for an environment that is empty except for the bindings for the standard procedures described in the previous revision of this report [5], omitting `load`, `transcript-on`, `transcript-off`, and `char-ready?`. The bindings have as values the procedures of the same names described in this report.

18. Sample definitions for derived forms

`case-lambda`

The `case-lambda` keyword (see section 13.2) could be defined in terms of base library by the following macros:

```
(define-syntax case-lambda
  (syntax-rules ()
    ((case-lambda
      (formals-0 body0-0 body1-0 ...)
      (formals-1 body0-1 body1-1 ...)
      ...)
     (lambda args
       (let ((l (length args)))
         (case-lambda-helper
          l args
          (formals-0 body0-0 body1-0 ...)
          (formals-1 body0-1 body1-1 ...) ...))))))

(define-syntax case-lambda-helper
  (syntax-rules ()
    ((case-lambda-helper
      l args
      ((formal ...) body ...)
      clause ...)
     (if (= l (length '(formal ...)))
         (apply (lambda (formal ...) body ...)
                args)
         (case-lambda-helper l args clause ...)))
    ((case-lambda-helper
      l args
      ((formal . formals-rest) body ...)
      clause ...)
     (case-lambda-helper-dotted
       l args
       (body ...)
       (formal . formals-rest)
       formals-rest 1
       clause ...))
    ((case-lambda-helper
      l args
      (formal body ...))
     (let ((formal args))
       body ...))))

(define-syntax case-lambda-helper-dotted
  (syntax-rules ()
    ((case-lambda-helper-dotted
      l args
      (body ...)
      formals
      (formal . formals-rest) k
      clause ...)
     (case-lambda-helper-dotted
      l args
```

```
    (body ...)
    formals
    formals-rest (+ 1 k)
    clause ...))
  ((case-lambda-helper-dotted
    l args
    (body ...)
    formals
    rest-formal k
    clause ...)
   (if (>= l k)
       (apply (lambda formals body ...) args)
       (case-lambda-helper
        l args clause ...)))))
```

REFERENCES

[1] Henk P. Barendregt. Introduction to the lambda calculus. *Nieuw Archief voor Wisenkunde*, 4(2):337–372, 1984.

[2] Alan Bawden. Quasiquotation in lisp. In Olivier Danvy, editor, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, pages 4–12, San Antonio, Texas, USA, January 1999. BRICS Notes Series NS-99-1.

[3] William Clinger and Jonathan Rees. Macros that work. In *Proc. 1991 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 155–162, Orlando, Florida, January 1991. ACM Press.

[4] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[5] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[6] Richard Kelsey and Michael Sperber. SRFI 34: Exception handling for programs. `http://srfi.schemers.org/srfi-34/`, 2002.

[7] Richard Kelsey and Michael Sperber. SRFI 35: Conditions. `http://srfi.schemers.org/srfi-35/`, 2002.

[8] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[9] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, and Jonathan Rees. Revised^6 report on the algorithmic language Scheme. `http://www.r6rs.org/`, 2007.

[10] The Unicode Consortium. The Unicode standard, version 5.0.0. defined by: *The Unicode Standard, Version 5.0* (Boston, MA, Addison-Wesley, 2007. ISBN 0-321-48091-0), 2007.

[11] Oscar Waddell. *Extending the Scope of Syntactic Abstraction*. PhD thesis, Indiana University, August 1999.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES