Revised5.95 Report on the Algorithmic Language Scheme
— Standard Libraries —

MICHAEL SPERBER
WILLIAM CLINGER, R. KENT DYBVIG, MATTHEW FLATT, ANTON VAN STRAATEN
(Editors)
RICHARD KELSEY, WILLIAM CLINGER, JONATHAN REES
(Editors, Revised5 Report on the Algorithmic Language Scheme)
24 June 2007

This report frequently refers back to the Revised6 Report on the Algorithmic Language Scheme; references to the report are identified by designations such as “report section” or “report chapter”.

Parts of the library report are derived from earlier revisions of the report. We gratefully acknowledge their authors for their contributions. More detailed information on authorship can be found at the beginning of the Revised6 Report on the Algorithmic Language Scheme.

We intend this report to belong to the entire Scheme community, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementors of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.

*** DRAFT***

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unicode</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Characters</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Strings</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Bytevectors</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Endianness</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>General operations</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Operations on bytes and octets</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Operations on integers of arbitrary size</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>Operations on 16-bit integers</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Operations on 32-bit integers</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Operations on 64-bit integers</td>
<td>8</td>
</tr>
<tr>
<td>2.8</td>
<td>Operations on IEEE-754 representations</td>
<td>9</td>
</tr>
<tr>
<td>2.9</td>
<td>Operations on strings</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>List utilities</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Sorting</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Control structures</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>Records</td>
<td>15</td>
</tr>
<tr>
<td>6.1</td>
<td>Mutability and equivalence</td>
<td>15</td>
</tr>
<tr>
<td>6.2</td>
<td>Procedural layer</td>
<td>16</td>
</tr>
<tr>
<td>6.3</td>
<td>Syntactic layer</td>
<td>19</td>
</tr>
<tr>
<td>6.4</td>
<td>Inspection</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Exceptions and conditions</td>
<td>22</td>
</tr>
<tr>
<td>7.1</td>
<td>Exceptions</td>
<td>22</td>
</tr>
<tr>
<td>7.2</td>
<td>Conditions</td>
<td>24</td>
</tr>
<tr>
<td>7.3</td>
<td>Standard condition types</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>I/O</td>
<td>27</td>
</tr>
<tr>
<td>8.1</td>
<td>Condition types</td>
<td>27</td>
</tr>
<tr>
<td>8.2</td>
<td>Port I/O</td>
<td>29</td>
</tr>
<tr>
<td>8.3</td>
<td>Simple I/O</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>File system</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>Command-line access and exit values</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>Arithmetic</td>
<td>41</td>
</tr>
<tr>
<td>11.1</td>
<td>Fixnums</td>
<td>41</td>
</tr>
<tr>
<td>11.2</td>
<td>Flonums</td>
<td>44</td>
</tr>
<tr>
<td>11.3</td>
<td>Exact bitwise arithmetic</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>syntax-case</td>
<td>48</td>
</tr>
<tr>
<td>12.1</td>
<td>Hygiene</td>
<td>48</td>
</tr>
<tr>
<td>12.2</td>
<td>Syntax objects</td>
<td>49</td>
</tr>
<tr>
<td>12.3</td>
<td>Transformers</td>
<td>49</td>
</tr>
<tr>
<td>12.4</td>
<td>Parsing input and producing output</td>
<td>49</td>
</tr>
<tr>
<td>12.5</td>
<td>Identifier predicates</td>
<td>51</td>
</tr>
<tr>
<td>12.6</td>
<td>Syntax-object and datum conversions</td>
<td>53</td>
</tr>
<tr>
<td>12.7</td>
<td>Generating lists of temporaries</td>
<td>54</td>
</tr>
<tr>
<td>12.8</td>
<td>Derived forms and procedures</td>
<td>54</td>
</tr>
<tr>
<td>12.9</td>
<td>Syntax violations</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Hashables</td>
<td>56</td>
</tr>
<tr>
<td>13.1</td>
<td>Constructors</td>
<td>56</td>
</tr>
<tr>
<td>13.2</td>
<td>Procedures</td>
<td>57</td>
</tr>
<tr>
<td>13.3</td>
<td>Inspection</td>
<td>57</td>
</tr>
<tr>
<td>13.4</td>
<td>Hash functions</td>
<td>58</td>
</tr>
<tr>
<td>14</td>
<td>Enumerations</td>
<td>58</td>
</tr>
<tr>
<td>15</td>
<td>Composite library</td>
<td>60</td>
</tr>
<tr>
<td>16</td>
<td>eval</td>
<td>60</td>
</tr>
<tr>
<td>17</td>
<td>Mutable pairs</td>
<td>60</td>
</tr>
<tr>
<td>18</td>
<td>Mutable strings</td>
<td>61</td>
</tr>
<tr>
<td>19</td>
<td>R5RS compatibility</td>
<td>61</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>13.5</td>
<td>Alphabetic index of definitions of concepts, keywords, and procedures</td>
<td>64</td>
</tr>
</tbody>
</table>
1. Unicode

The procedures exported by the (rnrs unicode (6)) 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. If the argument is not a title case character and there is no single character that is its title case form, then char-titlecase returns the upper case form of the argument. Finally, if the character has a casefolded character, then char-foldcase returns that character. Otherwise the character returned is the same as the argument. For Turkic characters I (#\x130) and I (#\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 #\$) \⇒ #\$3
(char-downcase #\$) \⇒ #\$3
(char-titlecase #\$) \⇒ #\$3
(char-foldcase #\$) \⇒ #\$3

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

(char-upcase #\ς) \⇒ #\ς
(char-downcase #\ς) \⇒ #\ς

Note: Note that char-titlecase does not always return a title case character.

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-titlecase #\ς) \⇒ #\Σ
(char-foldcase #\ς) \⇒ #\σ

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

(char-ci=? #\z #\Z) \⇒ #f
(char-ci=? #\z #\Σ) \⇒ #t
(char-ci=? #\ς #\σ) \⇒ #t

(char-alphabetic? char) procedure
(char-numeric? char) procedure
(char-whitespace? char) procedure
(char-upper-case? char) procedure
(char-lower-case? char) procedure
(char-title-case? char) 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 has the Unicode “Alphabetic” property. A character is numeric if it has the Unicode “Numeric” property. A character is whitespace if has the Unicode “White_Space” property. 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
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 lower case. 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 cased character of each word via char-titlecase, and downcases all other cased characters.

(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 "XAOΣ") ⇒ "XAOΣ"
(string-downcase "XAOΣ") ⇒ "χαος"
(string-downcase "XAOΣΣ") ⇒ "χαοςς"
(string-downcase "XAOΣΣΣ") ⇒ "χαος σ" (string-foldcase "XAOΣΣ") ⇒ "χαοςσ" (string-upcase "χαος") ⇒ "XAOΣ"
(string-upcase "χαος") ⇒ "XAOΣ"

(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.

Note: Word breaking, as needed for the correct casing of Σ and for string-titlecase, is specified in Unicode Standard Annex #29 [5].

(string-ci=? string1 string2 string3 ...) procedure
(string-ci<=? string1 string2 string3 ...) procedure
(string-ci>=? string1 string2 string3 ...) procedure
(string-ci>? string1 string2 string3 ...) procedure

These procedures are similar to string=? etc., 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=? "XAOΣ" "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.
2. Bytevectors

Many applications deal with blocks of binary data by accessing them in various ways—extracting signed and unsigned numbers of various sizes. Therefore, the (rnrs bytevector (6)) library provides a single type for blocks of binary data with multiple ways to access that data. It deals with integers and floating-point representations 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 object in the interval \([-128, \ldots, 127]\) and the term octet for an exact integer object 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 object. 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:

\[
\text{(string-normalize-nfd "\xE9;")} \quad \Rightarrow \quad \text{"\xE5;\x301;"} \\
\text{(string-normalize-nfc "\xE9;")} \quad \Rightarrow \quad \text{"\xE5;\x301;"} \\
\text{(string-normalize-nfd "\xE5;\x301;")} \quad \Rightarrow \quad \text{"\xE5;\x301;"} \\
\text{(string-normalize-nfc "\xE5;\x301;")} \quad \Rightarrow \quad \text{"\xE5;\x301;"} \\
\text{(string-normalize-nfc "\x65;\x301;")} \quad \Rightarrow \quad \text{"\x65;\x301;"} \\
\text{(string-normalize-nfc "\x65;\x301;")} \quad \Rightarrow \quad \text{"\x65;\x301;"} \,
\]

2.1. Endianness

Many operations described in this chapter accept an endianness argument. Endianness describes the encoding of exact integer objects as several contiguous bytes in a bytevector [4]. For this purpose, the binary representation of the integer object is split into consecutive bytes. The little-endian encoding places the least significant byte of an integer first, with the other bytes following in increasing order of significance. The big-endian encoding places the most significant byte of an integer first, with the other bytes following in decreasing order of significance.

This terminology also applies to IEEE-754 numbers: IEEE-754 describes how to represent a floating-point number as an exact integer object, and endianness describes how the bytes of such an integer are laid out in a bytevector.

Note: Little- and big-endianness are only the most common kinds of endianness. Some architectures distinguish between the endianness at different levels of a binary representation.

2.2. General operations

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

Note: Implementors are encouraged to use widely accepted designations for endianness symbols other than \text{big} and \text{little}.

\[(\text{native-endianness})\]  
\[\text{procedure}\]

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

\[(\text{bytevector? } obj)\]  
\[\text{procedure}\]

Returns \#t if \text{obj} is a bytevector, otherwise returns \#f.

\[(\text{make-bytevector } k)\]  
\[\text{procedure}\]

Returns a newly allocated bytevector of \text{k} bytes.

If the \text{fill} argument is missing, the initial contents of the returned bytevector are unspecified.

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

\[(\text{bytevector-length } bytevector)\]  
\[\text{procedure}\]

Returns, as an exact integer object, the number of bytes in \text{bytevector}.\]
(bytevector=? bytevector1 bytevector2) procedure
Returns #t if bytevector1 and bytevector2 are equal—that is, if they have the same length and equal bytes at all valid indices. It returns #f otherwise.

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

(bytevector-copy! source source-start procedure target target-start k)
Source and target must be bytevectors. Source-start, target-start, and k must be non-negative exact integer objects that satisfy
\[
0 \leq \text{source-start} \leq \text{source-start} + k \leq l_{\text{source}} \\
0 \leq \text{target-start} \leq \text{target-start} + k \leq l_{\text{target}}
\]
where \(l_{\text{source}}\) is the length of source and \(l_{\text{target}}\) is the length of target.
The bytevector-copy! procedure copies the bytes from source at indices
\[
\{\text{source-start}, \ldots, \text{source-start} + k - 1\}
\]
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 unspecified values.

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

(bytevector-copy bytevector) procedure
Returns a newly allocated copy of bytevector.

2.3. Operations on bytes and octets

(bytevector-u8-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))) \Rightarrow (-127 129 -1 255)

(bytevector-u8-set! bytevector k octet) 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 unspecified values.

(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))) \Rightarrow (-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 octets 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.

2.4. Operations on integers of arbitrary size

(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 object. \(\{k, \ldots, k + \text{size} - 1\}\) must be valid indices of bytevector.
The **bytevector-uint-ref** procedure retrieves the exact integer object corresponding to the unsigned representation of size `size` and specified by `endianness` at indices `{k, . . . , k + size - 1}`.

The **bytevector-sint-ref** procedure retrieves the exact integer object corresponding to the two’s complement representation of size `size` and specified by `endianness` at indices `{k, . . . , k + size - 1}`.

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

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

For **bytevector-sint-set!**, `n` must be an exact integer object in the interval `{−256^{size}/2, . . . , 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, . . . , k + size - 1}`.

The `...-set!` procedures return unspecified values.

```
(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) ➞ #xfffffffffffffffffffffffffffffff

(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)

(bytevector-uint-ref b 0 (endianness big) 16)

(bytevector-sint-ref b 0 (endianness big) 16) ➞ #xfffffffffffffffffffffffffffffff

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

2.5. Operations on 16-bit integers

The `...-ref` procedures return one-byte representations.

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

(bytevector-s16-ref bytevector k endianness)

(bytevector-u16-native-ref bytevector k)

(bytevector-s16-native-ref bytevector k)

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

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

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

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

`K` must be a valid index of `bytevector`; so must `k + 1`. For `bytevector-u16-set!` and `bytevector-u16-native-set!`, `n` must be an exact integer object in the interval `{0, . . . , 2^{16} − 1}`. For `bytevector-s16-set!` and `bytevector-s16-native-set!`, `n` must be an exact integer object in the interval `{−2^{15}, . . . , 2^{15} − 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 \texttt{s16} in their names deal with the two's complement representation.

The procedures with \texttt{native} in their names employ the native endianness, and work only at aligned indices: \(k\) must be a multiple of 2.

The \ldots\texttt{-set!} procedures return unspecified values.

\begin{verbatim}
(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-u16-ref b 14 (endianness little))
⇒ -513

(bytevector-s16-ref b 14 (endianness big))
⇒ 66533

(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
\end{verbatim}

2.6. Operations on 32-bit integers

\begin{verbatim}
(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-u16-ref b 14 (endianness little))
⇒ -513

(bytevector-s16-ref b 14 (endianness big))
⇒ 66533

(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
\end{verbatim}

2.7. Operations on 64-bit integers

\begin{verbatim}
(define b
  (u8-list->bytevector '
    (255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 253)))

<bytevector-u32-ref bytevector 12 (endianness little))
⇒ 4261412863

<bytevector-s32-ref bytevector 12 (endianness little))
⇒ -33554433

<bytevector-u32-ref bytevector 12 (endianness big))
⇒ 4294967293

<bytevector-s32-ref bytevector 12 (endianness big))
⇒ -3
\end{verbatim}

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

The procedures with \texttt{native} in their names employ the native endianness, and work only at aligned indices: \(k\) must be a multiple of 4.

The \ldots\texttt{-set!} procedures return unspecified values.

\begin{verbatim}
(define b
  (u8-list->bytevector '
    (255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 253)))

<bytevector-u32-ref bytevector 12 (endianness little))
⇒ 4261412863

<bytevector-s32-ref bytevector 12 (endianness little))
⇒ -33554433

<bytevector-u32-ref bytevector 12 (endianness big))
⇒ 4294967293

<bytevector-s32-ref bytevector 12 (endianness big))
⇒ -3
\end{verbatim}

{\(k, \ldots, k+7\)} must be valid indices of \texttt{bytevector}. For \texttt{bytevector-u32-set!} and \texttt{bytevector-s32-set!}, \(n\) must be an exact integer object in the interval \(\{-2^{31}, \ldots, 2^{31} - 1\}\). For \texttt{bytevector-s32-set!} and \texttt{bytevector-s32-native-set!}, \(n\) must be an exact integer object in the interval \(\{-2^{31}, \ldots, 2^{31} - 1\}\). For \texttt{bytevector-u32-native-set!}, \(n\) must be an exact integer object in the interval \(\{-2^{31}, \ldots, 2^{32} - 1\}\).
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 \(-\text{set!}\) procedures return unspecified values.

\[
\text{(define b (u8-list->bytevector '(255 255 255 255 255 255 255 255 255 255 255 255 255 255 253)))}
\]
\[
\text{(bytevector-u64-ref b 8 (endianness little)) = 18302628885633695743}
\]
\[
\text{(bytevector-s64-ref b 8 (endianness little)) = -144115188075855873}
\]
\[
\text{(bytevector-u64-ref b 8 (endianness big)) = 18446744073709551613}
\]
\[
\text{(bytevector-s64-ref b 8 (endianness big)) = -3}
\]

2.8. Operations on IEEE-754 representations

\[
\text{(bytevector-ieee-single-native-ref bytevector k)} \quad \text{procedure}
\]
\[
\text{(bytevector-ieee-single-ref bytevector k endianness)} \quad \text{procedure}
\]
\{\( k, \ldots, k + 3 \}\} must be valid indices of bytevector. For bytevector-ieee-single-native-set!, \( k \) must be a multiple of 4.

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

\[
\text{(bytevector-ieee-double-native-ref bytevector k)} \quad \text{procedure}
\]
\[
\text{(bytevector-ieee-double-ref bytevector k endianness)} \quad \text{procedure}
\]
\{\( 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 return the inexact real number object that best represents the IEEE-754 double precision representation of \( x \) into elements \( k \) through \( k + 7 \) of bytevector, and return unspecified values.

2.9. Operations on strings

This section describes procedures that convert between strings and bytevectors containing Unicode encodings of those strings. When decoding bytevectors, encoding errors are handled as with the \text{replace} semantics of textual \text{I/O} (see section [8.2.4]: If an invalid or incomplete character encoding is encountered, then the replacement character U+FFFD is appended to the string being generated, an appropriate number of bytes are ignored, and decoding continues with the following bytes.

\[
\text{(string->utf8 string)} \quad \text{procedure}
\]

Returns a newly allocated (unless empty) bytevector that contains the UTF-8 encoding of the given string.

\[
\text{(string->utf16 string endianness)} \quad \text{procedure}
\][8.2.4]

If endianness is specified, it must be the symbol big or the symbol little. The \text{string->utf16} \text{procedure} returns a newly allocated (unless empty) bytevector that contains the UTF-16BE or UTF-16LE encoding of the given string (with no byte-order mark). If endianness is not specified or is big, then UTF-16BE is used. If endianness is little, then UTF-16LE is used.

\[
\text{(string->utf32 string endianness)} \quad \text{procedure}
\][8.2.4]

If endianness is specified, it must be the symbol big or the symbol little. The \text{string->utf32} \text{procedure} returns a newly allocated (unless empty) bytevector that contains the UTF-32BE or UTF-32LE encoding of the given string (with no
byte mark). If endianness is not specified or is big, then UTF-32BE is used. If endianness is little, then UTF-32LE is used.

**(utf8->string bytevector)** procedure

Returns a newly allocated (unless empty) string whose character sequence is encoded by the given bytevector.

**(utf16->string bytevector endianness)** procedure

**(utf16->string bytevector endianness endianness-mandatory?)** procedure

Endianness is specified must be the symbol big or the symbol little. The `utf16->string` procedure returns a newly allocated (unless empty) string whose character sequence is encoded by the given bytevector. Bytevector is decoded according to UTF-16BE or UTF-16LE: If `endianness-mandatory?` is absent or `#f`, `utf16->string` determines the endianness according to a UTF-16 BOM at the beginning of `bytevector` if a BOM is present; in this case, the BOM is not decoded as a character. Also in this case, if no UTF-16 BOM is present, `endianness` specifies the endianness of the encoding. If `endianness-mandatory?` is a true value, `endianness` specifies the endianness of the encoding, and any UTF-16 BOM in the encoding is decoded as a regular character.

Note: A UTF-16 BOM is either a sequence of bytes `#xF0 #xF0` specifying big and UTF-16BE, or `#x00 #x00` specifying little and UTF-16LE.

**(utf32->string bytevector endianness)** procedure

**(utf32->string bytevector endianness endianness-mandatory?)** procedure

Endianness is specified must be the symbol big or the symbol little. The `utf32->string` procedure returns a newly allocated (unless empty) string whose character sequence is encoded by the given bytevector. Bytevector is decoded according to UTF-32BE or UTF-32LE: If `endianness-mandatory?` is absent or `#f`, `utf32->string` determines the endianness according to a UTF-32 BOM at the beginning of `bytevector` if a BOM is present; in this case, the BOM is not decoded as a character. Also in this case, if no UTF-32 BOM is present, `endianness` specifies the endianness of the encoding. If `endianness-mandatory?` is a true value, `endianness` specifies the endianness of the encoding, and any UTF-32 BOM in the encoding is decoded as a regular character.

Note: A UTF-32 BOM is either a sequence of bytes `#xF0 #xF0`, `#xF0 #x00` specifying big and UTF-32BE, or `#x00 #xF0`, `#x00 #x00` specifying little and UTF-32LE.

3. List utilities

This chapter describes the `(rnrs lists (6))` library.

(find proc list) procedure

Proc should accept one argument and return a single value. Proc should 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, `find` returns `#f`. Proc is always called in the same dynamic environment as `find` itself.

(find even? '(3 1 4 1 5 9)) ⇒ 4
(find even? '(3 1 4 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 list1 list2 ... listn) procedure

(exists proc list1 list2 ... listn) procedure

The lists should all have the same length, and proc should accept n arguments and return a single value. Proc should not mutate the list arguments.

For natural numbers i = 0, 1, …., the `for-all` procedure successively applies proc to arguments x1i , x2i, …, where x1i is the ith element of listj, until `#f` is returned. If proc returns true values for all but the last element of listj, `for-all` performs a tail call of proc on the kth elements, where k is the length of listj. If proc returns `#f` on any set of elements, `for-all` returns `#f` after the first such application of proc. If the lists are all empty, `for-all` returns `#t`.

For natural numbers i = 0, 1, …., the `exists` procedure successively to arguments x1i , x2i, …, where x1i is the ith element of listj, until a true value is returned. If proc returns `#f` for all but the last elements of the lists, `exists` performs a tail call of proc on the kth elements, where k is the length of listj. If proc returns a true value on any set of elements, `exists` returns that value after the first such application of proc. If the lists are all empty, `exists` returns `#f`.

Proc is always called in the 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
earlier returns are not mutated.

there are

\( \text{fold-left} \) combine \( \text{nil} \) list \( \text{proc} \) list

\( \text{filter} \) proc list

\( \text{partition} \) proc list

Proc should accept one argument and return a single value. Proc should not mutate list. The \( \text{filter} \) procedure applies proc to each element of list and returns a list of the elements of list for which proc returned a true value. The \( \text{partition} \) procedure also applies proc to each element of list, but returns two values, the first one a list of the elements of list for which proc returned a true value, and the second a list of the elements of list for which proc returned \#f. In both cases, the elements of the result list(s) are in the same order as they appear in the input list. Proc is always called in the same dynamic environment as \( \text{filter} \) or, respectively, \( \text{partition} \) itself. If multiple returns occur from \( \text{filter} \) or \( \text{partition} \), the return values returned by earlier returns are not mutated.

\( \text{filter even?} \) \( '(3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6) \) 

\( \Rightarrow (4 \ 2 \ 6) \)

\( \text{partition even?} \) \( '(3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6) \)

\( \Rightarrow (4 \ 2 \ 6) \ (3 \ 1 \ 1 \ 5 \ 9) \); two values

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

\( \text{fold-left} \) combine \( \text{nil} \) list \( \text{list}_1 \) \( \text{list}_2 \) \( \ldots \) \( \text{list}_n \)

The lists should all have the same length. \( \text{Combine} \) must be a procedure. It should accept one more argument than there are lists and return a single value. It should not mutate the list arguments. The \( \text{fold-left} \) procedure iterates the combine procedure over the elements of the lists from left to right, starting with an accumulator value of \text{nil}. More specifically, \( \text{fold-left} \) returns \text{nil} if the lists are empty. If they are not empty, combine is first applied to \text{nil} and the respective first elements of the lists in order. The result becomes the new accumulator value, and combine is applied to the 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. \( \text{Combine} \) is always called in the same dynamic environment as \( \text{fold-left} \) itself.

\( \text{fold-left} + \) \( '(1 \ 2 \ 3 \ 4) \) \( \Rightarrow 15 \)

\( \text{fold-left} \) \( \lambda (a \ e) \ (\text{cons} \ e \ a) \) \( '(1 \ 2 \ 3 \ 4 \ 5) \)

\( \Rightarrow (5 \ 4 \ 3 \ 2 \ 1) \)

\( \text{fold-left} \) \( \lambda (\text{x}) \ (\text{if} \ (\text{odd?} \ x) \ (+ \ \text{x} \ 1) \ \text{x})) \)

\( \Rightarrow 7 \)

\( \text{fold-left} \) \( \lambda (\text{s}) \ (\text{max} \ \text{max-len} \ (\text{string-length} \ \text{s})) \)

\( \Rightarrow 7 \)

\( \text{fold-left} \) \( \lambda (\text{a} \ b) \ (\text{cons} \ a \ b) \)

\( \Rightarrow (((q) . a) . b) . c) \)

\( \Rightarrow 21 \)

Implementation responsibilities: The implementation should check that the lists all have the same length. The implementation must check the restrictions on \( \text{combine} \) to the extent performed by applying it as described.
the list is reached; then the accumulator value is returned. Proc is always called in the same dynamic environment as fold-right itself.

\[ (\text{fold-right } + 0 \ '(1 \ 2 \ 3 \ 4 \ 5)) \Rightarrow 15 \]

\[ (\text{fold-right } \text{cons } '() '(1 \ 2 \ 3 \ 4 \ 5)) \Rightarrow (1 \ 2 \ 3 \ 4 \ 5) \]

\[ (\text{fold-right } (\lambda (x \ 1) \ (\text{if} \ (\text{odd?} \ x) \ (\text{cons} \ x \ 1)) \ ') \ '(3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ 5)) \Rightarrow (3 \ 1 \ 1 \ 5 \ 9 \ 5) \]

\[ (\text{fold-right } \text{cons } '(q)'(a \ b \ c)) \Rightarrow (a \ b \ c \ q) \]

\[ (\text{fold-right } + 0 \ '(1 \ 2 \ 3) '(4 \ 5 \ 6)) \Rightarrow 21 \]

**Implementation responsibilities:** The implementation should check that the lists all have the same length. The implementation must check the restrictions on combine to the extent performed by applying it as described.

\[ (\text{remp proc list}) \]
\[ (\text{remove obj list}) \]
\[ (\text{remq obj list}) \]
\[ (\text{memp proc list}) \]
\[ (\text{member obj list}) \]
\[ (\text{memv obj list}) \]
\[ (\text{memq obj list}) \]

Proc should accept one argument and return a single value. Proc should 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 the 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?.

\[ (\text{remp even? } '(3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ 5)) \Rightarrow (4 \ 1 \ 5 \ 9 \ 2 \ 6 \ 5) \]

\[ (\text{memq 'a '(a b c)) \Rightarrow (a b c) \]
\[ (\text{memq 'b '(a b c)) \Rightarrow (b c) \]
\[ (\text{memq 'a '(b c d)) \Rightarrow #f \]
\[ (\text{memq (list 'a) '(b (a) c)) \Rightarrow #f \]
\[ (\text{member (list 'a) '(b (a) c)) \Rightarrow (a c) \]
\[ (\text{memq 101 1'(100 101 102)) \Rightarrow \text{unspecified} \]
\[ (\text{memv 101 1'(100 101 102)) \Rightarrow (101 102) \]

**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.

\[ (\text{assp proc alist}) \]
\[ (\text{assoc obj alist}) \]
\[ (\text{assv obj alist}) \]
\[ (\text{assq obj alist}) \]

Alist (for “association list”) should be a list of pairs. Proc should accept one argument and return a single value. Proc should not mutate alist.

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
This chapter describes the (rnrs sorting (6)) library for sorting lists and vectors.

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

Proc should accept any two elements of the list or vector, and should not have any side effects. Proc should return 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 in ascending order according to proc, 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 proc where \(n\) is the length of list or vector, and all arguments passed to proc are elements of the list or vector being sorted, but the pairing of arguments and the sequencing of calls to proc are not specified.

(list-sort < '⟨3 5 2 1⟩) ⇒ ⟨1 2 3 5⟩
(vector-sort < '⟨#(3 5 2 1)⟩) ⇒ #⟨1 2 3 5⟩

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

(vector-sort! proc vector) procedure

Proc should accept any two elements of the vector, and should not have any side effects. Proc should return a true value when its first argument is strictly less than its second, and #f otherwise.

The vector-sort! procedure destructively sorts vector in ascending order according to proc. The sorting algorithm performs \(O(n^2)\) calls to proc where \(n\) is the length of vector, and all arguments passed to proc are elements of the vector being sorted, but the pairing of arguments and the sequencing of calls to proc are not specified. The sorting algorithm may be unstable. The procedure returns unspecified values.

(define v ⟨vector 3 5 2 1⟩)
(vector-sort! v) ⇒ unspecified
v ⇒ #⟨1 2 3 5⟩

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

5. Control structures

This chapter describes the (rnrs control (6)) library.

(when (test) ⟨expression1⟩ ⟨expression2⟩ . . . ) syntax
(unless (test) ⟨expression1⟩ ⟨expression2⟩ . . . ) 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 results of the last ⟨expression⟩ are returned as the results of the entire when expression. Otherwise, the when expression
returns unspecified values. An unless expression is evaluated by evaluating the (test) expression. If (test) evaluates to #f, the remaining (expression)s are evaluated in order, and the results of the last (expression) are returned as the results of the entire unless expression. Otherwise, the unless expression returns unspecified values.

$$\begin{align*}
&(\text{when} \ (> \ 3 \ 2) \ \text{`greater}) \ \Rightarrow \ \text{greater} \\
&(\text{when} \ (< \ 3 \ 2) \ \text{`greater}) \ \Rightarrow \ \text{unspecified} \\
&(\text{unless} \ (> \ 3 \ 2) \ \text{`less}) \ \Rightarrow \ \text{unspecified} \\
&(\text{unless} \ (< \ 3 \ 2) \ \text{`less}) \ \Rightarrow \ \text{less}
\end{align*}$$

The when and unless expressions are derived forms. They could be defined in terms of base library forms by the following macros:

```scheme
(define-syntax when
  (syntax-rules ()
    ((when test result1 result2 ...) (begin result1 result2 ...))))
(define-syntax unless
  (syntax-rules ()
    ((unless test result1 result2 ...) (begin result1 result2 ...))))
```

```scheme
(do ((variable1) (init1) (step1)) ...
    (test) (expression) ...
    (command) ...)
```

Syntax: The (init)s, (step)s, (test)s, and (command)s must be expressions. The (variable)s must be pairwise distinct variables.

Semantics: The do expression is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration.

A do expression is evaluated as follows: The (init) expressions are evaluated (in some unspecified order), the (variable)s are bound to fresh locations, the results of the (init) expressions are stored in the bindings of the (variable)s, and then the iteration phase begins.

Each iteration begins by evaluating (test); if the result is #f, then the (command)s are evaluated in order for effect, the (step) expressions are evaluated in some unspecified order, the (variable)s are bound to fresh locations holding the results, and the next iteration begins.

If (test) evaluates to a true value, the (expression)s are evaluated from left to right and the values of the last (expression) are returned. If no (expression)s are present, then the values of the do expression are unspecified.

The region of the binding of a (variable) consists of the entire do expression except for the (init)s. It is a syntax violation for a (variable) to appear more than once in the list of do variables.

A (step) may be omitted, in which case the effect is the same as if (variable) (init) (variable) had been written instead of (variable) (init).

If a do expression appears in a tail context, the (expression)s are a (tail sequence) in the sense of report section 9.21 i.e., the last (expression) is also in a tail context.

```scheme
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    (begin result1 result2 ...)))
```

The following definition of do uses a trick to expand the variable clauses.

```scheme
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...) (test expr ...) (command ...) ...) (letrec ((loop (lambda (var ...) (if test
        (begin
          #f ; avoid empty begin
          expr ...)
        (begin
          command...
        (loop (do "step" var step ...)
          ...) ...))))))
    (case-lambda (clause) ...)
```

Syntax: Each (clause) must be of the form

```scheme
((formals) (body))
```

(Formals) must be as in a lambda form (report section 9.5.2), and (body) is as described in 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 of 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⟩ . . . ⟨variable⟩⟩. . . ⟨⟨variable⟩⟩ 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 ()

The case-lambda keyword can be defined in terms of lambda by the following macros:

(define-syntax case-lambda
  (syntax-rules ()
    ((, (fmls b1 b2 ...) )
      (lambda fmls b1 b2 ...))
    ((, (fmls b1 b2 ...) ...)
      (lambda args
        (let ((n (length args)))
          (case-lambda-help args n
            (fmls b1 b2 ...) ...))))))

(define-syntax case-lambda-help
  (syntax-rules ()
    ((, args n)
      (assertion-violation #f
        "unexpected number of arguments")
    ((, args n ((x ... ) b1 b2 ...) more ...) (if (= n (length '(x ... )))
      (apply (lambda (x ... ) b1 b2 ... ) args)
      (case-lambda-help args n more ...)))
    ((, args n ((x1 x2 . . . r) b1 b2 ...) more ...) (if (> = n (length '(x1 x2 . . . )))
      (apply (lambda (x1 x2 . . . r) b1 b2 ... )
        args)
      (case-lambda-help args n more ...)))
    ((, args n (r b1 b2 ...) more ...) (apply (lambda r b1 b2 ... ) args))))

6. Records

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

• the (rnrs records procedural (6)) library, a procedural layer for creating and manipulating record types and record instances;

• the (rnrs records syntactic (6)) library, a syntactic layer for defining record types and various procedures to manipulate the record type; and

• the (rnrs records inspection (6)) 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 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 constructor, a predicate, field accessors, and field mutators. The layer allows the programmer to name each of these procedures explicitly, but also provides shorthands for naming them implicitly through a set of naming conventions.

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 facility 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 5.2).

6.1. Mutability and equivalence

The fields of a record type are designated mutable or immutable. Correspondingly, a record type with no mutable
field is called immutable, and all records of that type are immutable objects. All other record types are mutable, and so are their records.

Each call to a record constructor returns a new record with a fresh location (see report section 4.8). Consequently, for two records \texttt{obj}_1 and \texttt{obj}_2, the return value of \texttt{(eqv? obj}_1 \texttt{ obj}_2), adheres to the following criteria (see report section 9.6):

- If \texttt{obj}_1 and \texttt{obj}_2 have different record types (i.e., their record-type descriptors are not \texttt{eqv?}), \texttt{eqv?} returns \texttt{#f}.
- If \texttt{obj}_1 and \texttt{obj}_2 are both records of the same record type, and are the results of two separate calls to record constructors, then \texttt{eqv?} returns \texttt{#t}.
- If \texttt{obj}_1 and \texttt{obj}_2 are both records of the same record type, and both are the result of a single call to a record constructor, then \texttt{eqv?} returns \texttt{#f}.
- If \texttt{obj}_1 and \texttt{obj}_2 are both records of the same record type, where applying the same accessor to both yields results for which \texttt{eqv?} returns \texttt{#f}.

6.2. Procedural layer

The procedural layer is provided by the (\texttt{rnrs records procedural (6)}) library.

\texttt{(make-record-type-descriptor name \hspace{1em} procedure})
\texttt{parent \hspace{1em} uid \hspace{1em} sealed? \hspace{1em} opaque? \hspace{1em} fields)}

Returns a record-type descriptor, or \texttt{rtd}, representing a record type distinct from all built-in types and other record types.

The name argument must be a symbol. It names the record type, and is intended purely for informational purposes and may be used for printing by the underlying Scheme system.

The parent argument must be either \texttt{#f} or an rtd. If it is an rtd, the returned record type, \texttt{t}, extends the record type \texttt{p} represented by parent. Each record of type \texttt{t} is also a record of type \texttt{p}, and all operations applicable to a record of type \texttt{p} are also applicable to a record of type \texttt{t}, except for inspection operations if \texttt{t} is opaque but \texttt{p} is not. An exception with condition type \texttt{$\&assertion$} 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. A record type that does not extend another record type is called a base record type.

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

If \texttt{make-record-type-descriptor} is called twice with the same \texttt{uid} symbol, the parent arguments in the two calls must be \texttt{eqv?}, the \texttt{fields} arguments \texttt{equal?}, the \texttt{sealed?} arguments boolean-equivalent (both \texttt{#f} or both true), and the \texttt{opaque?} arguments boolean-equivalent. If these conditions are not met, an exception with condition type \texttt{$\&assertion$} 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 \texttt{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 \texttt{uid} argument.

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

The \texttt{opaque?} flag must be a boolean. If true, the record type is opaque. If passed an instance of the record type, \texttt{record?} returns \texttt{#f}. Moreover, if \texttt{record-rtd} (see “Inspection” below) is called an instance of the record type, an exception with condition type \texttt{$\&assertion$} is raised. The record type is also opaque if an opaque parent is supplied. If \texttt{opaque?} is \texttt{#f} and an opaque parent is not supplied, the record is not opaque.

The \texttt{fields} argument must be a vector of field specifiers. Each field specifier must be a list of the form (\texttt{mutable name}) or a list of the form (\texttt{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, when a program attempts to obtain a mutator for a field identified as immutable, an exception with condition type \texttt{$\&assertion$} is raised. 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 \texttt{fields} is modified after \texttt{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 \texttt{make-record-type-descriptor} is not \texttt{eqv?} to any record-type descriptor (generative or nongenerative) created by
another call to \texttt{make-record-type-descriptor}. A generative record-type descriptor is \texttt{eqv?} only to itself, i.e., \((\texttt{eqv? \ rtd_1 \ rtd_2})\) iff \((\texttt{eqv? \ rtd_1 \ rtd_2})\). Also, two nongenerative record-type descriptors are \texttt{eqv?} iff they were created by calls to \texttt{make-record-type-descriptor} with the same uid arguments.

\begin{verbatim}
(record-type-descriptor? obj) procedure
\end{verbatim}

Returns \texttt{#t} if the argument is a record-type descriptor, \texttt{#f} otherwise.

\begin{verbatim}
(make-record-constructor-descriptor rtd procedure parent-constructor-descriptor protocol)
\end{verbatim}

Returns a record-constructor descriptor (or constructor descriptor for short) that specifies a record constructor (or constructor for short), that can be used to construct record values of the type specified by \texttt{rtd}, and which can be obtained via \texttt{record-constructor}. A constructor descriptor can also be used to create other constructor descriptors for subtypes of its own record type. \texttt{Rtd} must be a record-type descriptor. \texttt{Protocol} must be a procedure or \texttt{#f}. If it is \texttt{#f}, a default \texttt{protocol} procedure is supplied.

If \texttt{protocol} is a procedure, it is called by \texttt{record-constructor} with a single argument \texttt{p} and should return a procedure that creates and returns an instance of the record type using \texttt{p} as described below.

The role of \texttt{p} differs depending on the kind of record type represented by \texttt{rtd}:

If \texttt{rtd} is a base record type, then \texttt{parent-constructor-descriptor} must be \texttt{#f}. In this case, \texttt{protocol}’s argument \texttt{p} is a procedure that expects one argument for every field of \texttt{rtd} and returns a record with the fields of \texttt{rtd} initialized to these arguments. The procedure returned by \texttt{protocol} should call \texttt{p} once with the number of arguments it expects and return the resulting record as shown in the simple example below:

\begin{verbatim}
(lambda (p)
 (lambda (v1 v2 v3)
 (p v1 v2 v3)))
\end{verbatim}

Here, the call to \texttt{p} returns a record whose fields are initialized with the values of \texttt{v1}, \texttt{v2}, and \texttt{v3}. The expression above is equivalent to \((\lambda (p) p)\). Note that the procedure returned by \texttt{protocol} is otherwise unconstrained; specifically, it can take any number of arguments.

If \texttt{rtd} is an extension of another record type \texttt{parent-rtd}, \texttt{parent-constructor-descriptor} must be a constructor descriptor of \texttt{parent-rtd} or \texttt{#f}. If \texttt{parent-constructor-descriptor} or \texttt{protocol} is \texttt{#f}, \texttt{protocol} must also be \texttt{#f}, and a default constructor descriptor is assumed as described below.

If \texttt{parent-constructor-descriptor} is a constructor descriptor and \texttt{protocol} is a procedure, then its argument \texttt{p} is a procedure that accepts the same number of arguments as the constructor of \texttt{parent-constructor-descriptor} and returns a procedure \texttt{new} that, when called, constructs the record itself. The \texttt{new} procedure expects one argument for every field of \texttt{rtd} (not including parent fields) and returns a record with the fields of \texttt{rtd} initialized to these arguments, and the fields of \texttt{parent-rtd} and its parents initialized as specified by \texttt{parent-constructor-descriptor}.

The procedure returned by \texttt{protocol} should call \texttt{p} once with the number of arguments it expects, call the procedure it returns once with number of arguments it expects and return the resulting record. A simple \texttt{protocol} in this case might be written as follows:

\begin{verbatim}
(lambda (p)
 (lambda (v1 v2 v3 x1 x2 x3 x4)
 (let ((new (p v1 v2 v3))
 (new x1 x2 x3 x4)))
\end{verbatim}

This passes arguments \texttt{v1}, \texttt{v2}, \texttt{v3} to \texttt{p} for \texttt{parent-constructor-descriptor} and calls \texttt{new} with \texttt{x1}, \ldots, \texttt{x4} to initialize the fields of \texttt{rtd} itself.

Thus, the constructor descriptors for a record type form a sequence of protocols exactly parallel to the sequence 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.

\texttt{Protocol} may be \texttt{#f}, specifying a default value that accepts one argument for each field of \texttt{rtd} (not including the fields of its parent type, if any). Specifically, if \texttt{rtd} is a base type, the default \texttt{protocol} procedure behaves as if it were \((\lambda (p) p)\). If \texttt{rtd} is an extension of another type, then \texttt{parent-constructor-descriptor} must be either \texttt{#f} or itself specify a default constructor. In this case, the default \texttt{protocol} procedure behaves as if it were:

\begin{verbatim}
(lambda (p)
 (lambda (v1 ... vj x1 ... xk)
 (let ((new (p v1 ... vj)))
 (new x1 ... xk))))
\end{verbatim}

The resulting constructor 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. (In the example, \texttt{j} is the complete number of fields of the parent type, and \texttt{k} is the number of fields of \texttt{rtd} itself.)

Implementation responsibilities: If \texttt{protocol} is a procedure, the implementation must check the restrictions on it to the extent performed by applying it as described when the constructor is called.

\begin{verbatim}
(record-constructor constructor-descriptor) procedure
\end{verbatim}

Calls the \texttt{protocol} of \texttt{constructor-descriptor} (as described for \texttt{make-record-constructor-descriptor}) and returns
the resulting constructor \textit{constructor} for records of the record type associated with \textit{constructor-descriptor}.

\texttt{(record-predicate \( \textit{rtd} \)) \hspace{1cm} \textit{procedure}}

Returns a procedure that, given an object \textit{obj}, returns a boolean that is \#t iff \textit{obj} is a record of the type represented by \textit{rtd}.

\texttt{(record-accessor \( \textit{rtd} \ \textit{k} \)) \hspace{1cm} \textit{procedure}}

\( \textit{K} \) must be a valid field index of \textit{rtd}. The \texttt{record-accessor} procedure returns a one-argument procedure that, given a record of the type represented by \textit{rtd}, returns the value of the selected field of that record.

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

If the accessor procedure is given something other than a record of the type represented by \textit{rtd}, an exception with condition type \texttt{assertion} is raised. Records of the type represented by \textit{rtd} include records of extensions of the type represented by \textit{rtd}.

\texttt{(record-mutator \( \textit{rtd} \ \textit{k} \)) \hspace{1cm} \textit{procedure}}

\( \textit{K} \) must be a valid field index of \textit{rtd}. The \texttt{record-mutator} procedure returns a two-argument procedure that, given a record \textit{r} of the type represented by \textit{rtd} and an object \textit{obj}, stores \textit{obj} within the field of \textit{r} specified by \textit{k}. The \textit{k} argument is as in \texttt{record-accessor}. If \textit{k} specifies an immutable field, an exception with condition type \texttt{assertion} is raised. The mutator returns unspecified values.

\texttt{(define :point \( \texttt{(make-record-type-descriptor \ 'point \ #f \ #f \ #f \ #f \ '#((mutable x) (mutable y)))} \))}

\texttt{(define :point-cd \( \texttt{(make-record-constructor-descriptor :point \ #f \ #f)} \))}

\texttt{(define make-point \( \texttt{(record-constructor :point-cd)} \))}

\texttt{(define point? \( \texttt{(record-predicate :point)} \))}

\texttt{(define point-x \( \texttt{(record-accessor :point \ 0)} \))}

\texttt{(define point-y \( \texttt{(record-accessor :point \ 1)} \))}

\texttt{(define point-x-set! \( \texttt{(record-mutator :point \ 0)} \))}

\texttt{(define point-y-set! \( \texttt{(record-mutator :point \ 1)} \))}

\texttt{(define p1 \( \texttt{(make-point \ 1 \ 2)} \))}

\texttt{(point? p1) \hspace{1cm} \#t}

\texttt{(point-x p1) \hspace{1cm} 1}

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

\texttt{(define make-make-point2 \( \texttt{(record-constructor \ (make-record-constructor-descriptor :point2 \ #f \ #f)} \))}

\texttt{(define point2? \( \texttt{(record-predicate :point2)} \))}

\texttt{(define point2-xx \( \texttt{(record-accessor :point2 \ 0)} \))}

\texttt{(define point2-yy \( \texttt{(record-accessor :point2 \ 1)} \))}

\texttt{(define p2 \( \texttt{(make-point2 \ 1 \ 2 \ 3 \ 4)} \))}

\texttt{(point? p2) \hspace{1cm} \#t}

\texttt{(point-x p2) \hspace{1cm} 1}

\texttt{(point-y p2) \hspace{1cm} 2}

\texttt{(point2-xx p2) \hspace{1cm} 3}

\texttt{(point2-yy p2) \hspace{1cm} 4}

\texttt{(define :point-cd/abs \( \texttt{(make-record-constructor-descriptor :point \ #f \ (lambda (new) \ (lambda (x y) \ ((new (abs x) (abs y))))))} \))}

\texttt{(define make-point-cd/abs \( \texttt{(record-constructor :point-cd/abs)} \))}

\texttt{(define point-x (make-point-cd/abs \ -1 \ -2))}

\texttt{(point-x (make-point-cd/abs \ -1 \ -2) \hspace{1cm} \#t}

\texttt{(point-y (make-point-cd/abs \ -1 \ -2) \hspace{1cm} 2}

\texttt{(define :cpoint \( \texttt{(make-record-type-descriptor \ 'cpoint \ :point \ #f \ #f \ #f \ '#((mutable rgb)))} \))}

\texttt{(define make-cpoint \( \texttt{(record-constructor \ (make-record-constructor-descriptor \ :cpoint \ :point-cd \ (lambda (p) \ (lambda (x y c) \ ((p x y) (color->rgb c))))))} \))}

\texttt{(define make-cpoint/abs \( \texttt{(record-constructor \ :point-cd/abs)} \))}

\texttt{(define point-x (make-point/abs \ -1 \ -2))}

\texttt{(point-x (make-point/abs \ -1 \ -2) \hspace{1cm} 1}

\texttt{(define :cpoint \( \texttt{(make-record-type-descriptor \ 'cpoint \ :point \ #f \ #f \ #f \ '#((mutable rgb)))} \))}

\texttt{(define make-cpoint \( \texttt{(record-constructor \ (make-record-constructor-descriptor \ :cpoint \ :point-cd \ (lambda (p) \ (lambda (x y c) \ ((p x y) (color->rgb c))))))} \))}

\texttt{(define make-cpoint/abs \( \texttt{(record-constructor \ :point-cd/abs)} \))}

\texttt{(define point-x (make-point/abs \ -1 \ -2))}

\texttt{(point-x (make-point/abs \ -1 \ -2) \hspace{1cm} 1}

\texttt{(define make-cpoint/abs \( \texttt{(record-constructor \ :point-cd/abs)} \))}

\texttt{(define point-x (make-point/abs \ -1 \ -2))}

\texttt{(point-x (make-point/abs \ -1 \ -2) \hspace{1cm} 1}
6. Records  19

6.3. Syntactic layer

The syntactic layer is provided by the \( \texttt{rnrs records syntactic (6)} \) library.

The record-type-defining form \texttt{define-record-type} is a definition and can appear anywhere any other \texttt{(definition)} can appear.

\[
\text{(define-record-type \texttt{name spec} \texttt{(record clause)*})}
\]

\textit{syntax}

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

The \texttt{(name spec)} specifies the names of the record type, constructor, and predicate. It must take one of the following forms:

\[
\text{(record name) (constructor name) (predicate name)
\texttt{ record name)}
\]

\texttt{(Record name), (constructor name), and (predicate name)} must all be identifiers.

\texttt{(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 \texttt{record-type-descriptor} and \texttt{record-constructor-descriptor} below).}

\texttt{(Constructor name) is defined by this definition to be a constructor for the defined record type, with a protocol specified by the \texttt{protocol clause}, or, in its absence, using a default protocol. For details, see the description of the \texttt{protocol clause} below.}

\texttt{(Predicate name) is defined by this definition to a predicate for the defined record type.}

The second form of \texttt{(name spec)} is an abbreviation for the first form, where the name of the constructor is generated by prefixing the record name with \texttt{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 \texttt{frob}, the name of the constructor is \texttt{make-frob}, and the predicate name is \texttt{frob?}.

Each \texttt{(record clause)} must take one of the following forms; it is a syntax violation if multiple \texttt{(record clause)s} of the same kind appear in a \texttt{define-record-type} form.

- \texttt{(fields (field spec)*)}

  where each \texttt{(field spec)} has one of the following forms

  - \texttt{(immutable \texttt{field name} \texttt{accessor name})}
  - \texttt{(mutable \texttt{field name})}
  - \texttt{(immutable \texttt{field name} \texttt{mutator name})}
  - \texttt{(mutable \texttt{field name})}
  - \texttt{(field name)}

  \texttt{(Field name), (accessor name), and (mutator name) must all be identifiers. The first form declares an immutable field called \texttt{field name}, with the corresponding accessor named \texttt{accessor name}. The second form declares a mutable field called \texttt{field name}, with the corresponding accessor named \texttt{accessor name}, and with the corresponding mutator named \texttt{mutator name}.}

  If \texttt{(field spec)} takes the second or third form, 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 \texttt{-set!} suffix to the accessor name. For example, if the record name is \texttt{frob} and the field name is \texttt{widget}, the accessor name is \texttt{frob-widget}, and the mutator name is \texttt{frob-widget-set!}.

  If \texttt{(field spec)} is just a \texttt{(field name)} form, it is an abbreviation for \texttt{(immutable \texttt{field name})}.

  The \texttt{(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.

  The \texttt{fields} clause may be absent; this is equivalent to an empty \texttt{fields} clause.

- \texttt{(parent \texttt{parent name})}

  Specifies that the record type is to have parent type \texttt{parent name}, where \texttt{parent name} is the

\[
\text{(define cpoint-rgb}
\text{(record-accessor :cpoint 0))}
\]

\[
\text{(define \texttt{(color->rgb c)}}
\text{(cons \texttt{‘rgb c))}}
\]

\[
\text{(cpoint-rgb \texttt{(make-cpoint -1 -3 ‘red)}}
\text{⇒ \texttt{(rgb . red)}}
\]

\[
\text{(point-x \texttt{(make-cpoint -1 -3 ‘red)}}
\text{⇒ \texttt{-1)}}
\]

\[
\text{(point-x \texttt{(make-cpoint/abs -1 -3 ‘red)}}
\text{⇒ \texttt{1)}}
\]
(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. If this option is specified with operand #f, or is absent, 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 constructor, the predicate, the accessors, and the mutators) must have names that are pairwise distinct.

The fields, mutable, immutable, parent, protocol, sealed, opaque, and nongenerative identifiers are all exported by the (rnrs records syntactic (6)) library with level 0. Referring to one of these identifiers out of place is a syntax violation.

Any definition that takes advantage of implicit naming for the constructor, predicate, accessor, and mutator names, can be rewritten trivially to a definition that specifies all names explicitly. For example, the implicit-naming record definition:

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

is equivalent to the following explicit-naming record definition.

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

Also, the implicit-naming record definition:

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

is equivalent to the following explicit-naming record definition:

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

With implicit naming, 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))))))
```
(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).

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

(define-record-type (cpoint make-cpoint cpoint?)
  (parent point)
  (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 p1 (make-point 1 2))
(define p2 (make-cpoint 3 4 'red))

(point? p1) ⇒ #t
(point? p2) ⇒ #t
(point? (vector)) ⇒ #f
(point? (cons 'a 'b)) ⇒ #f
(cpoint? p1) ⇒ #f
(cpoint? p2) ⇒ #t
(point-x p1) ⇒ 1
(point-y p1) ⇒ 2
(point-x p2) ⇒ 3
(point-y p2) ⇒ 4
(cpoint-rgb p2) ⇒ (rgb . red)

(set-point-y! p1 17)
(point-y p1) ⇒ 17)

(record-rtd p1)
⇒ (record-type-descriptor point)

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

(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

6.4. Inspection

The inspection layer is provided by the (rnrs records inspection (6)) 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.
On the one hand, 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 a record. 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.

7. 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 to dynamically establish 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.

7.1. Exceptions
This section describes Scheme’s exception-handling and exception-raising constructs provided by the (rnrs exceptions (6)) library.

Note: This specification follows SRFI 34 [7].

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 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; see report section [1.10]
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 and should accept one argument. Thunk must be a procedure that accepts zero arguments. The with-exception-handler procedure returns the results 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.

Implementation responsibilities: The implementation must check the restrictions on handler to the extent performed by applying it as described when it is called as a result of a call to raise or raise-continuable.

(guard ((variable) (cond clause1) (cond clause2) ... ) (body)) syntax
Syntax: Each (cond clause) is as in the specification of cond. (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 (cond clause)'s (test) evaluates to #f 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.

The => and else identifiers are exported from the (rnrs exceptions (6)) library with level 0, and are the same as in the (rnrs base (6)) library.

(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. When the handler returns, a non-continuable exception with condition type &non-continuable is raised in the same dynamic environment as the handler.

(raise-continuable obj) procedure
Raises a continueable 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) ((error? con) (display (condition-message con)) (display "an error has occurred"))) 'error) 
((violation? con) ((violation? 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) ((error? con) (display (condition-message con)) (display "an error has occurred"))) 'error))

(raise (condition (&violation) (&message (message "I am an error"))))
prints: &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))
  ...))
7.2. Conditions

The section describes Scheme's (rnrs conditions (6)) 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.

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.

7.2.1. Condition objects

Conceptually, there are two different kinds of condition objects: simple conditions and compound conditions. An object that is either a simple condition or a compound condition is simply a condition. Compound conditions form a type disjoint from the base types described in report section 9.2. A simple condition describes a single aspect of an exceptional situation. A compound condition represents multiple aspects of an exceptional situation as a list of simple conditions, its components. Most of the operations described in this section treat a simple condition identically to a compound condition consisting of only the simple condition. Thus, a simple condition is its own sole component. For a subtype t of &condition, a condition of type t is either a record of type t or a compound condition containing a component of type t.

&condition condition type

Simple conditions are records of subtypes of the &condition record type. The &condition type is neither sealed nor opaque.

```
((message-condition? con)
 (display (condition-message con)))
(else
 (display "a warning has been issued")))
(lambda ()
 (+ (raise-continuable
 (condition (&warning)
 (&message
 (message "should be a number"))))
 23)))
prints: should be a number
⇒ 65
```

The condition procedure returns a condition object with the components of the conditions as its components, in the same order, i.e., with the components of condition1 appearing first in the same order as in condition1, then with the components of condition2, and so on. The returned condition is compound if the total number of components is zero or greater than one. Otherwise, it may be compound or simple.

```
(simple-conditions condition)
```

The simple-conditions procedure returns a list of the components of condition, in the same order as they appeared in the construction of condition. The returned list is immutable. If the returned list is modified, the effect on condition is unspecified.

```
(condition? obj)
```

Returns #t if obj is a (simple or compound) condition, otherwise returns #f.

```
(condition-predicate rtd)
```

Rtd must be a record-type descriptor of a subtype of &condition. The condition-predicate procedure returns a procedure that takes one argument. This procedure returns #t if its argument is a condition of the condition type represented by rtd, i.e., if it is either a simple condition of that record type (or one of its subtypes) or a compound condition with such a simple condition as one of its components.

```
(condition-accessor rtd proc)
```

Rtd must be a record-type descriptor of a subtype of &condition. Proc should accept one argument, a record of the record type of rtd. The condition-accessor procedure returns a procedure that accepts a single argument, which must be a condition of the type represented by rtd. This procedure extracts the first component of the condition of the type represented by rtd, and returns the result of applying proc to that component.

```
(define-record-type (&cond1 make-cond1 real-cond1?)
 (parent &condition)
 (fields
  (immutable x real-cond1-x)))
(define cond1?
 (condition-predicate
  (record-type-descriptor &cond1)))
```
(define cond1-x
  (condition-accessor
   (record-type-descriptor &cond1)
   real-cond1-x))

(define foo (make-cond1 'foo))
(condition? foo) ⇒ #t
(cond1? foo) ⇒ #t
(cond1-x foo) ⇒ foo

(define-record-type (&cond2 make-cond2 real-cond2?)
  (parent &condition)
  (fields
   (immutable y real-cond2-y)))

(define cond2?
  (condition-predicate
   (record-type-descriptor &cond2)))

(define cond2-y
  (condition-accessor
   (record-type-descriptor &cond2)
   real-cond2-y))

(define bar (make-cond2 'bar))
(condition? (condition foo bar)) ⇒ #t
(cond1? (condition foo bar)) ⇒ #t
(cond2? (condition foo bar)) ⇒ #t
(real-cond1? (condition foo)) ⇒ unspecified
(real-cond2? (condition foo bar)) ⇒ #f
(cond1-x (condition foo bar)) ⇒ foo
(cond2-y (condition foo bar)) ⇒ bar

(equal? (simple-conditions (condition foo bar))) (list foo bar)) ⇒ #t

(equal? (simple-conditions (condition foo bar))) (list foo bar)) ⇒ #t

(define-condition-type ⟨condition-type⟩ syntax
  (supertype)
  ⟨constructor⟩ ⟨predicate⟩
  ⟨field-spec1⟩ ...)

Syntax: (Condition-type), (supertypes), (constructor), 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 record-type definition for a record type &condition-type (see section 6.3). The record type will be non-opaque, non-sealed, and its fields will be immutable. It will have ⟨supertype⟩ has its parent type. The remaining identifiers will be bound as follows:

- ⟨Constructor⟩ is bound to a default constructor for the type (see section 6.2): It accepts one argument for each of the record type’s complete set of fields (including parent types, with the fields of the parent coming before those of the extension in the arguments) and returns a condition object initialized to those arguments.

- ⟨Predicate⟩ is bound to a predicate that identifies conditions of type ⟨condition-type⟩ or any of its subtypes.

- Each ⟨accessor⟩ is bound to a procedure that extracts the corresponding field from a condition of type ⟨condition-type⟩.

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

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

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

(define v1 (make-c1 "V1" "a1"))
(c? v1) ⇒ #t
(c1? v1) ⇒ #t
(c2? v1) ⇒ #t
(c-x v1) ⇒ "V1"
(c1-a v1) ⇒ "a1"

(define v2 (make-c2 "V2" "b2"))
(c? v2) ⇒ #t
(c1? v2) ⇒ #t
(c2? v2) ⇒ #t
(c-x v2) ⇒ "V2"
(c2-b v2) ⇒ "b2"

(define v3 (condition
  (make-c1 "V3/1" "a3")
  (make-c2 "V3/2" "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 (condition v1 v2))

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

(define v5 (condition v2 v3))

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

7.3. Standard condition types

&message condition type
(make-message-condition message) procedure
(message-condition? obj) procedure
(condition-message condition) procedure

This condition type could be defined by

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

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

&warning condition type
(make-warning) procedure
(warning? obj) procedure

This condition type could be defined by

(define-condition-type &warning &condition
make-warning 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
(make-serious-condition) procedure
(serious-condition? obj) procedure

This condition type could be defined by

(define-condition-type &serious &condition
make-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
(make-error) procedure
(error? obj) procedure

This condition type could be defined by

(define-condition-type &error &serious
make-error 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
(make-violation) procedure
(violation? obj) procedure

This condition type could be defined by

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

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

&non-continuable condition type
(make-non-continuable-violation) procedure
(non-continuable-violation? obj) procedure

This condition type could be defined by

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

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

&implementation-restriction condition type
(make-implementation-restriction-violation) procedure
(implementation-restriction-violation? obj) procedure

This condition type could be defined by

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

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 11.2)

&lexical condition type
(make-lexical-violation) procedure
(lexical-violation? obj) procedure

This condition type could be defined by
This type describes syntax violations at the level of the read syntax.

\[
\text{(define-condition-type \&lexical \&violation}
\text{make-lexical-violation lexical-violation?)}
\]

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

\[
\text{\&syntax} \quad \text{condition type}
\text{(make-syntax-violation form subform)} \quad \text{procedure}
\text{(syntax-violation? obj)} \quad \text{procedure}
\text{(syntax-violation-form condition)} \quad \text{procedure}
\text{(syntax-violation-subform condition)} \quad \text{procedure}
\]

This condition type could be defined by

\[
\text{(define-condition-type \&syntax \&violation}
\text{make-syntax-violation syntax-violation?}
\text{(form syntax-violation-form)}
\text{(subform syntax-violation-subform))}
\]

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.

\[
\text{\&undefined} \quad \text{condition type}
\text{(make-undefined-violation)} \quad \text{procedure}
\text{(undefined-violation? obj)} \quad \text{procedure}
\]

This condition type could be defined by

\[
\text{(define-condition-type \&undefined \&violation}
\text{make-undefined-violation undefined-violation?)}
\]

This type describes unbound identifiers in the program.

\[
\text{\&assertion} \quad \text{condition type}
\text{(make-assertion-violation)} \quad \text{procedure}
\text{(assertion-violation? obj)} \quad \text{procedure}
\]

This condition type could be defined by

\[
\text{(define-condition-type \&assertion \&violation}
\text{make-assertion-violation assertion-violation?}
\text{(form assertion-violation-form)}
\text{(subform assertion-violation-subform))}
\]

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

\[
\text{\&irritants} \quad \text{condition type}
\text{(make-irritants-condition irritants)} \quad \text{procedure}
\text{(irritants-condition? obj)} \quad \text{procedure}
\text{(condition-irritants condition)} \quad \text{procedure}
\]

This condition type could be defined by

\[
\text{(define-condition-type \&irritants \&condition}
\text{make-irritants-condition irritants-condition?}
\text{(irritants condition-irritants))}
\]

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 9.15.

\[
\text{\&who} \quad \text{condition type}
\text{(make-who-condition who)} \quad \text{procedure}
\text{(who-condition? obj)} \quad \text{procedure}
\text{(condition-who condition)} \quad \text{procedure}
\]

This condition type could be defined by

\[
\text{(define-condition-type \&who \&condition}
\text{make-who-condition who-condition?}
\text{(who condition-who))}
\]

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 9.15), and the syntax-violation procedure (section 12.9).

8. I/O

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

- The (rnrs io ports (6)) library (section 8.2) is an I/O layer for conventional, imperative buffered input and output with mixed text and binary data.
- The (rnrs io simple (6)) library (section 8.3) is a convenience library atop the (rnrs io ports (6)) library for textual I/O, compatible with the traditional Scheme I/O procedures [6].

Section 8.1 defines a condition-type hierarchy that is exported by both the (rnrs io ports (6)) and (rnrs io simple (6)) libraries.

8.1. Condition types

The procedures described in this chapter, when they detect an exceptional situation that arises from an “I/O errors”, raise an exception with condition type \&i/o.

The condition types and corresponding predicates and accessors are exported by both the (rnrs io ports (6)) and (rnrs io simple (6)) libraries. They are also exported by the (rnrs files (6)) library described in chapter 9.

\[
\text{\&i/o} \quad \text{condition type}
\text{(make-i/o-error)} \quad \text{procedure}
\text{(i/o-error? obj)} \quad \text{procedure}
\]

This condition type could be defined by
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.

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

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

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.

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.

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

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.

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.

This condition type could be defined by

This condition type could be defined by

This condition type could be defined by

This condition type could be defined by

This condition type could be defined by
A condition of this type specifies that an operation tried to
operate on a non-existent named file under the assump-
tion that it existed.

This condition type could be defined by

\[
\text{(define-condition-type \&i/o-port \&i/o}
\text{make-i/o-port-error i/o-port-error? (port i/o-error-port))}
\]

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 pro-
cedures may include an \&i/o-port-error condition, but
are not required to do so.

8.2. Port I/O

The (rnrs io ports (6)) 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 (rnrs io ports (6)) 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 (rnrs io ports (6)) library also distinguishes be-
tween binary ports, which are sources or sinks for uninter-
preted 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.

8.2.1. 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 oper-
ating system, and may include implementation-dependent
values as well.

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

8.2.2. 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 14) 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
textual syntax returns a file-options object that en-
capsulates the specified options.

When supplied to an operation that opens a file for output,
the file-options object returned by (file-options) spec-
ifies that the file is created if it does not exist and an ex-
ception with condition type &i/o-file-already-exists
is raised if it does exist. The following standard options
can be included to modify the default behavior.

- no-create If the file does not already exist, it is
  not created; instead, an exception with condition
  type &i/o-file-does-not-exist is raised. If the
  file already exists, the exception with condition type
  &i/o-file-already-exists is not raised and the file
  is truncated to zero length.

- no-fail If the file already exists, the exception with
  condition type &i/o-file-already-exists is not
  raised, even if no-create is not included, and the file
  is truncated to zero length.

- no-truncate If the file already exists and the excep-
  tion with condition type &i/o-file-already-exists
  has been inhibited by inclusion of no-create or
  no-fail, the file is not truncated, but the port’s cur-
  rent position is still set to the beginning of the file.

These options have no effect when a file is opened only
for input. (Identifiers) other than those listed above may
be used as (file-options name)s; they have implementation-
specific meaning, if any.
8.2.3. Buffer modes

Each port has an associated buffer mode. For an output port, the buffer mode defines when an output operation flushes the buffer associated with the output port. For an input port, the buffer mode defines how much data will be read to satisfy read operations. The possible buffer modes are the symbols none for no buffering, line for flushing upon line endings or reading until line endings, and block for arbitrary buffering. This section uses the parameter name buffer-mode for arguments that must be buffer-mode symbols.

If two ports are connected to the same mutable source, both ports are unbuffered, and reading a byte or character from that shared source via one of the two ports would change the bytes or characters seen via the other port, a lookahead operation on one port will render the peeked byte or character inaccessible via the other port, while a subsequent read operation on the peeked port will see the peeked byte or character even though the port is otherwise unbuffered.

In other words, the semantics of buffering is defined in terms of side effects on shared mutable sources, and a lookahead operation has the same side effect on the shared source via one of the two ports would change the bytes or characters seen via the other port. If two ports are connected to the same mutable source, both ports are unbuffered, and reading a byte or character from that shared source via one of the two ports would change the bytes or characters seen via the other port, a lookahead operation on one port will render the peeked byte or character inaccessible via the other port, while a subsequent read operation on the peeked port will see the peeked byte or character even though the port is otherwise unbuffered.

In other words, the semantics of buffering is defined in terms of side effects on shared mutable sources, and a lookahead operation has the same side effect on the shared source as a read operation.

(buffer-mode (name))

(buffer-mode? obj)

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

8.2.4. Transcoders

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

An end-of-line style is a symbol that, if it is not none, describes how a textual port transcodes representations of line endings.

A transcoder is an immutable Scheme object that combines a codec with an end-of-line style 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 is not). The composition of two transcoders is not defined. A transcoder parameter name means that the corresponding argument must be a transcoder.

A binary port is a port that supports binary I/O, does not have an associated transcoder and does not support textual I/O. A textual port is a port that supports textual I/O, and does not support binary I/O. A textual port may or may not have an associated transcoder.

(latin-1-codec) (utf-8-codec) (utf-16-codec)

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

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)

If name is one of the (identifier)s lf, cr, crlf, nel, crnel, ls, or none, the form evaluates to the corresponding symbol. If name is not one of these identifiers, the effect and result are implementation-dependent; in particular, the result may be an eol-style symbol acceptable as an eol-mode argument to make-transcoder. Otherwise, an exception is raised.

All eol-style symbols except none describe a specific line-ending encoding:

lf (linefeed) cr (carriage return) crlf (carriage return) (linefeed) nel (next line) crnel (carriage return) (next line) ls (line separator)

For a textual port with a transcoder, and whose transcoder has an eol-style symbol none, no conversion occurs. For a textual input port, any eol-style symbol other than none means that all of the above line-ending encodings are recognized and are translated into a single linefeed. For a textual output port, none and lf are equivalent. Linefeed characters are encoded according to the specified eol-style symbol, and all other characters that participate in possible line endings are encoded as is.

(native-eol-style)

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

Implementation responsibilities: The implementation must check that the exception handler returns a character or a string only if it actually returns.

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, should output to the port an appropriate encoding for the character that caused the error. The operation that raised the exception continues after that character.
(native-transcoder)  procedure

Returns an implementation-dependent transcoder that represents a possibly locale-dependent "native" transcoding.

(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, and handling-mode arguments, respectively.

(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.

8.2.5. 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 the argument is a port, and returns #f otherwise.

(port-transcoder port)  procedure

Returns the transcoder associated with port if port is textual and has an associated transcoder, and returns #f if port is binary or does not have an associated transcoder.

(textual-port? port)  procedure
(binary-port? port)  procedure

The textual-port procedure returns #t if port is textual, and returns #f otherwise. The binary-port procedure returns #t if port is binary, and returns #f otherwise.

(transcoded-port binary-port transcoder)  procedure

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, 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, 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, transcoded-port 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.

(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.

For a binary port, the port-position procedure returns the index of the position at which the next byte would be read from or written to the port as an exact non-negative integer object. For a textual port, port-position returns an arbitrary value that is acceptable as input for set-port-position! (see below).

If the port does not support the operation, port-position raises an exception with condition type &assertion.

Note: For a textual port, the port position may or may not be an integer object. If it is an integer object, the integer object does not necessarily correspond to a byte or character position.
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 raises an exception with condition type &assertion if the port does not support the operation. Otherwise, it sets the current position of the port to pos. If port is an output port, set-port-position! first flushes port. (See flush-output-port, section 8.2.10)

If port is a binary output port and the current position is set beyond the current end of the data in the underlying data sink, the object is not extended until new data is written at that position. The contents of any intervening positions are unspecified. Binary ports created by open-file-output-port and open-bytevector-output-port can always be extended in this manner within the limits of the underlying operating system. In other cases, attempts to set the port beyond the current end of data in the underlying object may result in an exception with condition type &io-invalid-position.

The call-with-port procedure returns unspecified values.

Proc must accept one argument. The call-with-port procedure calls proc with port as an argument. If proc returns, the port is closed automatically and the values returned by proc are returned. If proc does not return, the port is not closed automatically, except perhaps when it is possible to prove that the port will never again be used for an input or output operation.

### 8.2.7. Input ports

An input port allows the reading of 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 &io-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.

The open-bytevector-input-port procedure returns an input port whose bytes are drawn from the bytevector. If

Maybe-transcoder must be either a transcoder or #f.

The open-bytevector-input-port 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 maybe-transcoder is #f or absent, 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 may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent. 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 a fresh binary input port connected to standard input. Whether the port supports the port-position and set-port-position! operations is implementation-dependent.

(current-input-port) procedure

This returns a default textual port for input. Normally, this default port is associated with standard input, but can be dynamically re-assigned using the with-input-from-file procedure from the (rnrs io simple (6)) library (see section 8.3). The port may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent.

Implementation responsibilities: The implementation is required to check the return values of read! and get-position only when it actually calls them as part of an I/O operation requested by the program. The implementation is not required to check that these procedures otherwise behave as described. If they do not, however, the behavior of the resulting port is unspecified.

(make-custom-binary-input-port id read! procedure 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 object, count will be a positive exact integer object, and bytevector will be a bytevector whose length is at least start + count. The read! procedure should obtain up to count bytes from the byte source, and should write those bytes into bytevector starting at index start. The read! procedure should return an exact integer object. This integer object should represent the number of bytes that it has read. To indicate an end of file condition, the read! procedure should write no bytes and return 0.

• (get-position)

The get-position procedure (if supplied) should return an exact integer object. The return value should represent the current position of the input port. If not supplied, the custom port will not support the port-position operation.

• (set-position! pos)

Pos will be a non-negative exact integer object. The set-position! procedure (if supplied) should set the position of the input port to pos. If not supplied, the custom port will not support the set-port-position! operation.

• (close)

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

(make-custom-textual-input-port id read! procedure get-position set-position! close) procedure

Returns a newly created textual input port whose character 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 textual 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.
8.2.8. Binary input

(read! string start count)  
Start will be a non-negative exact integer object, count will be a positive exact integer object, and string will be a string whose length is at least start + count. The read! procedure should obtain up to count characters from the character source, and should write those characters into string starting at index start. The read! procedure must return an exact integer object. This integer object should represent the number of characters that it has written. To indicate an end of file condition, the read! procedure should write no bytes and return 0.

(get-position)  
The get-position procedure (if supplied) should return a single value. The return value should represent the current position of the input port. If not supplied, the custom port will not support the port-position operation.

(set-position! pos)  
The set-position! procedure (if supplied) should set the position of the input port to pos if pos is the return value of a call to get-position. If not supplied, the custom port will not support the set-port-position! operation.

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

The port may or may not have an an associated transcoder; if it does, the transcoder is implementation-dependent.

Implementation responsibilities: The implementation is required to check the return values of read! and get-position only when it actually calls them as part of an I/O operation requested by the program. The implementation is not required to check that these procedures otherwise behave as described. If they do not, however, the behavior of the resulting port is unspecified.

8.2.8. Binary input

(lookahead-u8 binary-input-port)  
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 count)  
Count must be an exact, non-negative integer object representing the number of bytes to be read. 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 bytes are available before an end of file, get-bytevector-n returns a bytevector of size count. 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)  
Count must be an exact, non-negative integer object, representing 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 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 a number object representing 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)  
Proceeds 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. The operation may block indefinitely waiting to see if more data will become available, even if some bytes are already available.

**8.2.9. Textual input**

```scheme
(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, `get-char` returns the end-of-file object.

```scheme
(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.

```scheme
(get-string-n textual-input-port count) procedure
```

*Count* must be an exact, non-negative integer object, representing the number of characters to be read.

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

If `count` characters are available before end of file, `get-string-n` returns a string consisting of those `count` 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, the end-of-file object is returned.

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

`Start` and `count` must be exact, non-negative integer objects, with `count` representing 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` 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, those characters are written into string starting at index `start` and the number of characters actually read is returned as an exact integer object. If no characters can be read before an end of file, the end-of-file object is returned.

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

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

Reads from `textual-input-port` up to and including the linefeed character or end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

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

**Note:** The end-of-line style, if not `none`, will cause all line endings to be read as linefeed characters. See section 8.2.5.

```scheme
(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 8.2.5) 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.

8.2.10. Output ports

An output port is a sink to which bytes or characters are written. The written 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), #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 flush-output-port procedure returns unspecified values.

(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 maybe-transcoder) procedure
Maybe-transcoder must be either a transcoder or #f.

The open-file-output-port procedure returns an output port for the named file. The file-options argument, which may determine various aspects of the returned port (see section 8.2.2), defaults to the value of (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 maybe-transcoder is a transcoder, it becomes the transcoder associated with the port.

If maybe-transcoder is #f or absent, 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).

(open-bytevector-output-port maybe-transcoder) procedure
(open-bytevector-output-port maybe-transcoder) procedure
Maybe-transcoder must be either a transcoder or #f.

The open-bytevector-output-port procedure returns two values: an output port and an extraction procedure. The output port accumulates the data written to it for later extraction by the procedure.

If maybe-transcoder is a transcoder, it becomes the transcoder associated with the port. If maybe-transcoder is #f or absent, 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).

The extraction procedure takes no arguments. When called, it returns a bytevector consisting of all the port’s accumulated data (regardless of the port’s current position), removes the accumulated data from the port, and resets the port’s position.

(call-with-bytevector-output-port proc maybe-transcoder) procedure
Proc must accept one argument. Maybe-transcoder must be either a transcoder or #f.

The call-with-bytevector-output-port procedure 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 all of the port’s accumulated data (regardless of the port’s current position) is returned and the port is closed.

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

(open-string-output-port) procedure
Returns two values: a textual output port and an extraction procedure. The output port accumulates the characters written to it for later extraction by the procedure.

The port may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent. The port should support the port-position and set-port-position! operations.

The extraction procedure takes no arguments. When called, it returns a string consisting of all of the port’s accumulated characters (regardless of the current position), removes the accumulated characters from the port, and resets the port’s position.
Procedure \(\text{proc}\) must accept one argument. Creates a textual output port that accumulates the characters written to it and calls \(\text{proc}\) with that output port as an argument. Whenever \(\text{proc}\) returns, a string consisting of all of the port’s accumulated characters (regardless of the port’s current position) is returned and the port is closed.

The port may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent. The port should support the \text{port-position} and \text{set-port-position!} operations.

\[\text{(standard-output-port)}\quad \text{(standard-error-port)}\]

Returns a fresh binary output port connected to the standard output or standard error respectively. Whether the port supports the \text{port-position} and \text{set-port-position!} operations is implementation-dependent.

\[\text{(current-output-port)}\quad \text{(current-error-port)}\]

These return default textual ports for regular output and error output. Normally, these default ports are associated with standard output, and standard error, respectively. The return value of \text{current-output-port} can be dynamically re-assigned using the \text{with-output-to-file} procedure from the \text{(rnrs io simple (6))} library (see section 8.3). A port returned by one of these procedures may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent.

\[\text{(make-custom-binary-output-port id \text{write! get-position set-position! close)}}\]

Returns a newly created binary output port whose byte sink is an arbitrary algorithm represented by the \text{write!} procedure. \(\text{id}\) must be a string naming the new port, provided for informational purposes only. \text{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 \text{#f}; if any of those arguments is not \text{#f}, it must be a procedure and should behave as specified in the description of \text{make-custom-binary-input-port}.

- (\text{write! bytevector start count})
  
  \(\text{Start}\) and \(\text{count}\) will be non-negative exact integer objects, and \text{bytevector} will be a bytevector whose length is at least \(\text{start} + \text{count}\). The \text{write!} procedure should read up to \(\text{count}\) bytes from \text{bytevector} starting at index \(\text{start}\) and forward them to the byte sink. If \(\text{count}\) is 0, the \text{write!} procedure should have the effect of passing an end-of-file object to the byte sink. In any case, the \text{write!} procedure should return the number of bytes that it wrote, as an exact integer object.

\[\text{Implementation responsibilities:}\quad \text{The implementation is required to check the return values of write! only when it actually calls write! as part of an I/O operation requested by the program. The implementation is not required to check that write! otherwise behaves as described. If it does not, however, the behavior of the resulting port is unspecified.}\]

\[\text{(make-custom-textual-output-port id \text{write! get-position set-position! close)}}\]

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

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

- (\text{write! string start count})

  \(\text{Start}\) and \(\text{count}\) will be non-negative exact integer objects, and \text{string} will be a string whose length is at least \(\text{start} + \text{count}\). The \text{write!} procedure should read up to \(\text{count}\) characters from \text{string} starting at index \(\text{start}\) and forward them to the character sink. If \(\text{count}\) is 0, the \text{write!} procedure should have the effect of passing an end-of-file object to the character sink. In any case, the \text{write!} procedure should return the number of characters that it wrote, as an exact integer object.

\[\text{Implementation responsibilities:}\quad \text{The implementation is required to check the return values of write! only when it actually calls write! as part of an I/O operation requested by the program. The implementation is not required to check that write! otherwise behaves as described. If it does not, however, the behavior of the resulting port is unspecified.}\]

### 8.2.11. Binary output

\[\text{(put-u8 binary-output-port octet)}\quad \text{procedure}\]

Writes \text{octet} to the output port and returns unspecified values.
(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 integer objects that default to 0 and (bytevector-length bytevector) – start, respectively. bytevector must have a length of at least start + count. The put-bytevector procedure writes the count bytes of the bytevector bytevector starting at index start to the output port. The put-bytevector procedure returns unspecified values.

8.2.12. Textual output

(put-char textual-output-port char) procedure
 Writes char to the port. The put-char procedure returns unspecified values.

(put-string textual-output-port string) procedure
(put-string textual-output-port string start count) procedure
Start and count must be non-negative exact integer objects. 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 put-string procedure returns unspecified values.

(make-custom-textual-input/output-port procedure id read! write! get-position set-position! close) procedure
Returns a newly created textual input/output port whose textual 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.

(make-custom-textual-input/output-port procedure id read! write! get-position set-position! close) procedure
Returns a newly created input/output port whose textual 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.

8.3. Simple I/O

This section describes the (rnrs io simple (6)) 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 [8].

The ports created by the procedures of this library are textual ports associated implementation-dependent transcoders.
(eof-object) procedure
(eof-object? obj) procedure
These are the same as eof-object and eof-object? from the (rnrs ports (6)) library.

(call-with-input-file filename proc) procedure
(call-with-output-file filename proc) procedure
Proc should accept one 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, the port is closed automatically and the values returned by proc are returned. If proc does not return, 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 (rnrs io ports (6)) library.

(current-input-port) procedure
(current-output-port) procedure
(current-error-port) procedure
These are the same as the current-input-port, current-output-port, and current-error-port procedures from the (rnrs io ports (6)) library.

(with-input-from-file filename thunk) procedure
(with-output-to-file filename thunk) procedure
Thunk must be a procedure and should accept zero 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, 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 8.2.9
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. This returns unspecified values.
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 \texttt{write-char} instead of by \texttt{write}. The \texttt{display} procedure returns unspecified values. The \texttt{textual-output-port} argument may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.

\begin{verbatim}
(write obj) \hspace{1em} procedure
(write obj textual-output-port) \hspace{1em} procedure
\end{verbatim}

Writes the external representation of \texttt{obj} to \texttt{textual-output-port}. The \texttt{write} procedure operates in the same way as \texttt{put-datum}; see section \ref{sec:put-datum}.

If \texttt{textual-output-port} is omitted, it defaults to the value returned by \texttt{current-output-port}.

\section{File system}

This chapter describes the (rnrs files (6)) 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 \ref{sec:i/o-condition-types}.

\begin{verbatim}
(file-exists? filename) \hspace{1em} procedure
\end{verbatim}

\texttt{Filename} must be a filename (see section \ref{sec:filename}). The \texttt{file-exists?} procedure returns \texttt{#t} if the named file exists at the time the procedure is called, \texttt{#f} otherwise.

\begin{verbatim}
(delete-file filename) \hspace{1em} procedure
\end{verbatim}

\texttt{Filename} must be a filename (see section \ref{sec:filename}). The \texttt{delete-file} procedure deletes the named file if it exists and can be deleted, and returns unspecified values. If the file does not exist or cannot be deleted, an exception with condition type \texttt{&io-filename} is raised.

\section{Command-line access and exit values}

The procedures described in this section are exported by the (rnrs programs (6)) library.

\begin{verbatim}
(command-line) \hspace{1em} procedure
\end{verbatim}

Returns a nonempty list of strings. The first element is an implementation-specific name for the running top-level program. The remaining elements are command-line arguments according to the operating system’s conventions.

\begin{verbatim}
(exit) \hspace{1em} procedure
(exit obj) \hspace{1em} procedure
\end{verbatim}

Exits the running program and communicates an exit value to the operating system. If no argument is supplied, the \texttt{exit} procedure should communicate to the operating system that the program exited normally. If an argument is supplied, the \texttt{exit} procedure should translate the argument into an appropriate exit value for the operating system. If \texttt{obj} is \texttt{#f}, the exit is assumed to be abnormal.

\section{Arithmetic}

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

\subsection{Fixnums}

Every implementation must define its fixnum range as a closed interval

\[ [-2^w - 1, 2^w - 1] \]

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 object that is representable within the implementation. A fixnum is an exact integer object whose value lies within this fixnum range.

This section describes the (rnrs arithmetic fx (6)) library, which defines various operations on fixnums. Fixnum operations perform integer arithmetic on their fixnum arguments, but raise an exception with condition type \texttt{&implementation-restriction} if the result is not a fixnum.

This section uses \texttt{fx}, \texttt{fx1}, \texttt{fx2}, etc., as parameter names for arguments that must be fixnums.

\begin{verbatim}
(fixnum? obj) \hspace{1em} procedure
\end{verbatim}

Returns \texttt{#t} if \texttt{obj} is an exact integer object within the fixnum range, \texttt{#f} otherwise.

\begin{verbatim}
(fixnum-width) \hspace{1em} procedure
(least-fixnum) \hspace{1em} procedure
(greatest-fixnum) \hspace{1em} procedure
\end{verbatim}

These procedures return \( w \), \( -2^w - 1 \) and \( 2^w - 1 \): the width, minimum and the maximum value of the fixnum range, respectively.

\begin{verbatim}
(fx=? fx1 fx2 fx3 ...) \hspace{1em} procedure
(fx>? fx1 fx2 fx3 ...) \hspace{1em} procedure
(fx<=? fx1 fx2 fx3 ...) \hspace{1em} procedure
(fx>=? fx1 fx2 fx3 ...) \hspace{1em} procedure
(fx<=>? fx1 fx2 fx3 ...) \hspace{1em} procedure
\end{verbatim}

These procedures return \texttt{#t} if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, \texttt{#f} otherwise.
# Scheme Libraries

- (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 object is zero, greater than zero, less than zero, odd, or even.

- (fmax fx1 fx2 ...) procedure
- (fmin fx1 fx2 ...) procedure

These procedures return the maximum or minimum of their arguments.

- (fx+ fx1 fx2) procedure
- (fx* fx1 fx2) 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.

- (fx- fx1 fx2) 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 object 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`

- (fxdiv0-and-mod fx1 fx2) procedure
- (fxdiv fx1 fx2) procedure
- (fxmod fx1 fx2) procedure
- (fxdiv0-and-mod0 fx1 fx2) procedure
- (fxdiv0 fx1 fx2) procedure
- (fxmod0 fx1 fx2) procedure
- (fxdiv0-and-mod0 fx1 fx2) procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section 9.8.3.

- (fxdiv fx1 fx2) `⇒` `fx1 div fx2`
- (fxmod fx1 fx2) `⇒` `fx1 mod fx2`
- (fxdiv-and-mod fx1 fx2) `⇒` `fx1 div fx2`, `fx1 mod fx2`

- (fxmod0 fx1 fx2) `⇒` `fx1 mod fx2`
- (fxdiv0-and-mod0 fx1 fx2) `⇒` `fx1 div fx2`, `fx1 mod fx2`
- (fxdiv0 fx1 fx2) `⇒` `fx1 div fx2`, `fx1 mod fx2`
- (fxmod0 fx1 fx2) `⇒` `fx1 mod fx2`

- (fxdiv0-and-mod0 fx1 fx2) `⇒` `fx1 div fx2`, `fx1 mod fx2`

- (fx- (least-fixnum)) procedure

- (fx* fx1 fx2) procedure
- (fx* fx1 fx2) procedure

These procedures return the two fixnum results of the following computation:

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

- (fx/ carry fx1 fx2 fx3) procedure

Returns the two fixnum results of the following computation:

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

- (fx/ carry fx1 fx2 fx3) procedure

Returns the two fixnum results of the following computation:

- (let* ((s (+ (* s1 s2 fx3)))
  (s0 (mod0 s (expt 2 (fixnum-width))))
  (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))

- (fx/ carry fx1 fx2 fx3) procedure

Returns the two fixnum results of the following computation:

- (let* ((s (+ (+ s1 s2 fx3)))
  (s0 (mod0 s (expt 2 (fixnum-width))))
  (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))

- (fx/ carry fx1 fx2 fx3) procedure

Returns the two fixnum results of the following computation:

- (fxif fx1 fx2 fx3) procedure

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

- (fxand fx1 ...) procedure
- (fxior fx1 ...) procedure
- (fxxor fx1 ...) 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 fx1 fx2 fx3) procedure

Returns the fixnum result of the following computation:

- (fxif (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 result of the following computation:

\[(\text{fxnot} \ \text{fxbit-count} \ \text{fxnot} \ \text{ei})\]

(fxlength fx) procedure

Returns the fixnum result of the following computation:

\[(\text{do} \ \text{(result} \ 0 \ \text{(+ result} \ 1)) \ \text{bits} \ \text{(if} \ \text{fxnegative?} \ \text{fx} \ \text{fxnot} \ \text{fx}) \ \text{fx}) \ \text{(fxarithmetic-shift-right bits} \ 1))) \ \text{(fxzero? bits) result})\]

(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.

\[(\text{fxfirst-bit-set} \ 0) \Rightarrow -1 \ \text{(fxfirst-bit-set} \ 1) \Rightarrow 0 \ \text{(fxfirst-bit-set} \ -4) \Rightarrow 2\]

(fxbit-set? fx fx1 fx2) procedure

Fx2 must be non-negative and less than \((\text{fixnum-width})\). The \text{fxbit-set?} procedure returns the fixnum result of the following computation:

\[(\text{not} \ \text{(fxzero?} \ \text{fxand} \ \text{fx1} \ \text{fxarithmetic-shift-left} \ 1 \ \text{fx2}))\]

(fxcopy-bit fx1 fx2 fx3) procedure

Fx2 must be non-negative and less than \((\text{fixnum-width})\). Fx3 must be 0 or 1. The \text{fxcopy-bit} procedure returns the result of the following computation:

\[(\text{let*} \ ((\text{mask} \ \text{fxarithmetic-shift-left} \ 1 \ \text{fx2})) \ \text{fxif mask} \ \text{fxarithmetic-shift-left} \ \text{fx3} \ \text{fx2} \ \text{fx1})))\]

(fxbit-field fx1 fx2 fx3) procedure

Fx2 and fx3 must be non-negative and less than \((\text{fixnum-width})\). Moreover, fx2 must be less than or equal to fx3. The \text{fxbit-field} procedure returns the fixnum result of the following computation:

\[(\text{let*} \ ((\text{mask} \ \text{fxarithmetic-shift-left} \ -1 \ \text{fx3})) \ \text{fxarithmetic-shift-right} \ \text{fxand} \ \text{fx1} \ \text{mask} \ \text{fx2})))\]

(fxcopy-bit-field fx1 fx2 fx3 fx4) procedure

Fx2 and fx3 must be non-negative and less than \((\text{fixnum-width})\). Moreover, fx2 must be less than or equal to fx3. The \text{fxcopy-bit-field} procedure returns the fixnum result of the following computation:

\[(\text{let*} \ ((\text{to} \ \text{fx1}) \ \text{(start} \ \text{fx2}) \ \text{(end} \ \text{fx3}) \ \text{(mask1} \ \text{fxarithmetic-shift-left} \ -1 \ \text{start}) \ \text{mask2} \ \text{fxnot} \ \text{fxarithmetic-shift-left} \ -1 \ \text{end}) \ \text{mask} \ \text{fxand} \ \text{mask1} \ \text{mask2})\]

\[(\text{fxif mask} \ \text{fxarithmetic-shift-left from} \ \text{to})\]

(fxarithmetic-shift fx1 fx2) procedure

The absolute value of fx2 must be less than \((\text{fixnum-width})\). If

\[(\ast \ \text{fx1} \ \text{expt} \ 2 \ \text{fx2})\]

is a fixnum, then that fixnum is returned. Otherwise an exception with condition type \text{implementation-restriction} is raised.

(fxarithmetic-shift-left fx1 fx2) procedure

(fxarithmetic-shift-right fx1 fx2) procedure

Fx2 must be non-negative. \text{fxarithmetic-shift-left} behaves the same as \text{fxarithmetic-shift}, and \text{fxarithmetic-shift-right} behaves the same as \text{fxarithmetic-shift-left} \ ((\text{fixnum-width}) \ fx2)\).

(fxrotate-bit-field fx1 fx2 fx3) procedure

Fx2, fx3, and fx4 must be non-negative and less than \((\text{fixnum-width})\). Fx4 must be less than the difference between fx3 and fx2. The \text{fxrotate-bit-field} procedure returns the result of the following computation:

\[(\text{let*} \ ((\text{n} \ \text{fx0}) \ \text{(start} \ \text{fx2}) \ \text{(end} \ \text{fx3}) \ \text{(count} \ \text{fx4}) \ \text{(width} \ \text{fx-} \ \text{end} \ \text{start})) \ \text{fxif mask} \ \text{fxarithmetic-shift-left from} \ \text{to})\]

\[(\text{field0} \ \text{fxbit-field} \ \text{n} \ \text{start} \ \text{end} \ \text{field0} \ \text{fx-} \ \text{width} \ \text{count}) \ \text{field1} \ \text{fxarithmetic-shift-right field0} \ \text{fx-} \ \text{width} \ \text{count}) \ \text{field2} \ \text{fxarithmetic-shift-right field0} \ \text{fx-} \ \text{width} \ \text{count}) \ \text{fxcopy-bit-field} \ \text{n})\]
(fxreverse-bit-field fx1 fx2 fx3) procedure

Fx3 and fx2 must be non-negative and less than
(fixnum-width). Moreover, fx2 must be less than or equal
to fx3. The fxreverse-bit-field procedure returns the
fixnum obtained from fx2 by reversing the bit field specified
by fx2 and fx3.

(fxreverse-bit-field #b1010010 1 4)
⇒ 88 ; #b1011000
(fxreverse-bit-field #b1010010 91 -4)
⇒ 82 ; #b1010010

11.2. Flonums

This section describes the (rnrs arithmetic flonum
(6)) library.

This section uses fl, fl1, fl2, etc., as parameter names for
arguments that must be flonums, and ifl as a name 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, #f otherwise.

(real->flonum x) procedure

Returns the best flonum representation of x.
The value returned is a flonum that is numerically closest
to the argument.

Note: If flonums are represented in binary floating point, then
implementations are strongly encouraged to break ties by pre-
ferring the floating point representation whose least significant
bit is zero.

(fl=? fl1 fl2 fl3 ...) procedure
(fl<? fl1 fl2 fl3 ...) procedure
(fl<=? fl1 fl2 fl3 ...) procedure
(fl>? fl1 fl2 fl3 ...) procedure
(fl>=? fl1 fl2 fl3 ...) procedure

These procedures return #t if their arguments are (respec-
tively): equal, monotonically increasing, monotonically de-
creasing, 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) ⇒ #f
(fl= 0.0 -0.0) ⇒ #t
(fl< 0.0 -0.0) ⇒ #f
(fl< +nan.0 fl) ⇒ #f
(fl< +nan.0 fl) ⇒ #f

(fl+ fl1 ...) procedure
(fl* fl1 ...) 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
using IEEE binary floating point, the meaning of “best” is
defined by the IEEE standards.)

(fl+ +inf.0 -inf.0) ⇒ +nan.0
(fl+ +inf.0 0.0) ⇒ +inf.0
(fl+ +inf.0 0.0) ⇒ +inf.0

(fl- fl1 fl2 ...) procedure
(fl/ fl1 fl2 ...) 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 using IEEE binary floating point, the meaning of “best” is reasonably well-defined by the IEEE standards.)

\[
\begin{align*}
(fl- +inf.0 +inf.0) & \implies +nan.0 \\
\end{align*}
\]

For undefined quotients, \(fl/\) behaves as specified by the IEEE standards:

\[
\begin{align*}
(fl/ 1.0 0.0) & \implies +inf.0 \\
(fl/ -1.0 0.0) & \implies -inf.0 \\
(fl/ 0.0 0.0) & \implies +nan.0 \\
\end{align*}
\]

\[
(flabs fl)
\]

Returns the absolute value of \(fl\).

\[
\begin{align*}
(fldiv-and-mod fl_1 fl_2) & \implies fl_1 div fl_2, fl_1 mod fl_2 \\
(fldiv fl_1 fl_2) & \implies fl_1 div fl_2 \\
(flmod fl_1 fl_2) & \implies fl_1 mod fl_2 \\
(fldiv0-and-mod0 fl_1 fl_2) & \implies fl_1 div fl_2, fl_1 mod fl_2 \\
(fldiv0 fl_1 fl_2) & \implies fl_1 div0 fl_2 \\
(flmod0 fl_1 fl_2) & \implies fl_1 mod0 fl_2 \\
(fldiv0-and-mod0 fl_1 fl_2) & \implies fl_1 div0 fl_2, fl_1 mod0 fl_2 \\
\end{align*}
\]

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section 9.8.3. For zero divisors, these procedures may return a NaN or some meaningless flonum.

\[
\begin{align*}
(fldiv fl_1 fl_2) & \implies fl_1 div fl_2 \\
(flmod fl_1 fl_2) & \implies fl_1 mod fl_2 \\
(fldiv-and-mod fl_1 fl_2) & \implies fl_1 div fl_2, fl_1 mod fl_2 \\
(fldiv0 fl_1 fl_2) & \implies fl_1 div0 fl_2 \\
(flmod0 fl_1 fl_2) & \implies fl_1 mod0 fl_2 \\
(fldiv0-and-mod0 fl_1 fl_2) & \implies fl_1 div0 fl_2, fl_1 mod0 fl_2 \\
\end{align*}
\]

\[
\begin{align*}
(flnumerator fl) & \implies \text{flonum numerator of} fl \\
(fldenominator fl) & \implies \text{flonum denominator of} fl \\
\end{align*}
\]

The following behavior is strongly recommended but not required:

\[
\begin{align*}
(flnumerator -0.0) & \implies -0.0 \\
(flfloor fl) & \implies \text{flonum floor of} fl \\
(flceil fl) & \implies \text{flonum ceiling of} fl \\
(fltruncate fl) & \implies \text{flonum truncate of} fl \\
(flround fl) & \implies \text{flonum round of} fl \\
\end{align*}
\]

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 \(flceil\) 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 \(fl\) represents a number halfway between two integers.

Although infinities and NaNs are not integer objects, these procedures return an infinity when given an infinity as an argument, and a NaN when given a NaN:

\[
\begin{align*}
(flfloor +inf.0) & \implies +inf.0 \\
(flceil +inf.0) & \implies +inf.0 \\
(fltruncate +inf.0) & \implies +inf.0 \\
\end{align*}
\]

\[
\begin{align*}
(flexp fl) & \implies \text{flonum exponential of} fl \\
(filog fl) & \implies \text{flonum logarithm of} fl \\
(flsin fl) & \implies \text{flonum sine of} fl \\
(flcos fl) & \implies \text{flonum cosine of} fl \\
(fltan fl) & \implies \text{flonum tangent of} fl \\
(flasin fl) & \implies \text{flonum arc sine of} fl \\
(flacos fl) & \implies \text{flonum arc cosine of} fl \\
(flatan fl) & \implies \text{flonum arc tangent of} fl \\
(flatan fl_1 fl_2) & \implies \text{flonum arc tangent of} fl_1/ fl_2 \\
\end{align*}
\]

These procedures compute the usual transcendental functions. The \(flexp\) procedure computes the base-\(e\) exponential of \(fl\). The \(filog\) procedure with a single argument computes the natural logarithm of \(fl\) (not the base ten logarithm); \((filog fl_1 fl_2)\) 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 9.8.3 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.
11.3. Exact bitwise arithmetic

This section describes the (rnrs arithmetic bitwise (6)) library. The exact bitwise arithmetic provides generic operations on exact integer objects. This section uses \( ei, e1, e2, \ldots \), as parameter names that must be exact integer objects.

Some procedures allow extracting bit fields, i.e., number objects representing subsequences of the binary representation of an exact integer object. Bit fields are always positive, and always defined using a finite number of bits, contrary to 2’s complement representation which implicitly uses an infinite extension of 0 bits or 1 bits to the left.

(bitwise-not \( ei \)) procedure

Returns the exact integer object whose two’s complement representation is the one’s complement of the two’s complement representation of \( ei \).

(bitwise-and \( e1 \ldots \)) procedure

(bitwise-ior \( e1 \ldots \)) procedure

(bitwise-xor \( e1 \ldots \)) procedure

These procedures return the exact integer object 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 integer object (either −1 or 0) that acts as identity for the operation.

(bitwise-if \( e1 \ e2 \ e3 \)) procedure

Returns the exact integer object that is the result of the following computation:

\[
\text{(bitwise-ior (bitwise-and (bitwise-not \( e1 \)) \( e2 \)) (bitwise-and \( e1 \) (bitwise-not \( e3 \))))}
\]

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 result of the following computation:

\[
\text{(bitwise-not (bitwise-bit-count (bitwise-not \( ei \))))}
\]

(bitwise-length \( ei \)) procedure

Returns the exact integer object that is the result of the following computation:

\[
\text{(do ((result 0 (+ result 1)))
  (bits (if (negative? \( ei \))
    (bitwise-not \( ei \))
    \( ei \)))
  (bitwise-arithmetic-shift bits -1)))
\]

\( (\text{zero? bits result}) \)
(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) \(\Rightarrow -1\)
(bitwise-first-bit-set 1) \(\Rightarrow 0\)
(bitwise-first-bit-set -4) \(\Rightarrow 2\)

(bitwise-bit-set? ei ei2) procedure
Ei2 must be non-negative. Returns the result of the following computation:
(not (zero? 
  (bitwise-and 
    (bitwise-arithmetic-shift-left 1 ei2) 
    ei1)))

(bitwise-copy-bit ei1 ei2 ei3) procedure
Ei2 and ei3 must be non-negative, and ei2 must be less than or equal to ei3. This 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 ei1 ei2 ei3) procedure
Ei2 and ei3 must be non-negative, and ei2 must be less than or equal to ei3. 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 ei1 ei2 ei3 ei4) procedure
Ei2 and ei3 must be non-negative, and ei2 must be less than or equal to ei3. The bitwise-copy-bit-field procedure returns the result of the following computation:
(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)))
  (bitwise-copy-bit-field n start end field))

(bitwise-reverse-bit-field ei1 ei2 ei3) procedure
Ei2 and ei3 must be non-negative, and ei2 must be less than or equal to ei3. The bitwise-reverse-bit-field procedure returns the result obtained from ei1 by reversing the bit field specified by ei2 and ei3.

(bitwise-first-bit-field 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.

Examples:
(bitwise-arithmetic-shift -6 -1) \(\Rightarrow -3\)
(bitwise-arithmetic-shift -5 -1) \(\Rightarrow -3\)
(bitwise-arithmetic-shift -4 -1) \(\Rightarrow -2\)
(bitwise-arithmetic-shift -3 -1) \(\Rightarrow -2\)
(bitwise-arithmetic-shift -2 -1) \(\Rightarrow -1\)
(bitwise-arithmetic-shift -1 -1) \(\Rightarrow -1\)
Thus, the hygiene condition can be restated as follows:

\[
\begin{align*}
\text{bitwise-reverse-bit-field} & \ b1010010 \ 1 \ 4 \\
\Rightarrow & \ 88 \ ; b1011000 \\
\text{bitwise-reverse-bit-field} & \ b1010010 \ 91 \ -4 \\
\Rightarrow & \ \&\text{assertion} \ exception
\end{align*}
\]

12. syntax-case

The \(\text{rnrs syntax-case (6)}\) 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.

12.1. 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 \[12.4\]) inside the transformer body or one of the helpers it calls. This is essentially the referential transparency property described by Clinger and Rees \[9\].

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 \text{datum->syntax}; see section \[12.6\].

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 \text{lambda} expression, the expander might map each bound identifier to a representation of the formal parameter in the output of the expander. For a \text{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 \text{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 \text{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 \text{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 \text{eq?}), and the marks match if the marks recorded with the substitution are the same as those that appear \text{below} the substitution in the wrap, i.e., those that were applied \text{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].

12.2. 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 [12.1]) consists of a \texttt{wrap} (section [12.1]) 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 \texttt{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 may or may not be distinct from other types of values, but syntax objects representing identifiers 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:

\begin{itemize}
\item a pair of syntax objects,
\item a vector of syntax objects,
\item a nonpair, nonvector, nonsymbol value, or
\item a wrapped syntax object.
\end{itemize}

The distinction between the terms “syntax object” and “wrapped syntax object” is important. For example, when invoked by the expander, a transformer (section [12.3]) must accept a wrapped syntax object but may return any syntax object, including an unwrapped syntax object.

12.3. Transformers

In \texttt{define-syntax} (report section [9.3.2]), \texttt{let-syntax}, and \texttt{letrec-syntax} forms (report section [9.19]), a binding for a syntactic keyword must be an expression that evaluates to a \texttt{transformer}. (This is only the user’s responsibility; the implementation must check this only if evaluation of a transformer expression actually terminates. See the respective specifications.)

A transformer is a \texttt{transformation procedure} or a \texttt{variable transformer}. A transformation procedure is a procedure that must accept one argument, a wrapped syntax object (section [12.2]) representing the input, and return a syntax object (section [12.2]) representing the output. The transformer 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 car of a list-structured input form, the transformer receives the entire list-structured form, and its output replaces the entire form. Except with \texttt{variable} transformers (see below), 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 \texttt{variable} transformers, an exception with condition type \texttt{&syntax} is raised if the keyword appears on the left-hand side of a \texttt{set!} expression.

\begin{verbatim}
(make-variable-transformer proc) Procedure
Proc should accept one argument, a wrapped syntax object, and return a 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 \texttt{set!} expression, an exception is not raised. Instead, \texttt{proc} is called with a wrapped syntax object representing the entire \texttt{set!} expression as its argument, and its return value replaces the entire \texttt{set!} expression.

Implementation responsibilities: The implementation must check the restrictions on \texttt{proc} only to the extent performed by applying it as described.
\end{verbatim}

12.4. Parsing input and producing output

Transformers can destruct and rebuild their output with \texttt{syntax-case} and rebuild their output with \texttt{syntax}.

\begin{verbatim}
syntax-case \langle expression \rangle \langle (\text{literal} \ldots) \langle \text{clause} \ldots \rangle \rangle syntax
Syntax: Each \texttt{literal} must be an identifier. Each \texttt{clause} must take one of the following two forms.
\begin{verbatim}
\langle \text{pattern} \ (\text{output expression}) \rangle
\langle \text{pattern} \ \texttt{fender} \ (\text{output expression}) \rangle
\end{verbatim}
\end{verbatim}

\langle \text{Fender} \rangle \text{and} \ (\text{output expression}) \ must be \ (\text{expression})s.

A \langle \text{pattern} \rangle is an identifier, constant, or one of the following.
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 ((pattern) . . . ).

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 12.5).
- $P$ is of the form $(P_1 \ldots P_n)$ and $F$ is a list of $n$ elements that match $P_1$ through $P_n$.
- $P$ is of the form $(P_1 \ldots P_n . P_2)$ 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 $n$th cdr matches $P_2$.
- $P$ is of the form $(P_1 \ldots P_k P_e \ellipsis P_{m+1} \ldots P_n)$, 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 $(P_1 \ldots P_n)$ and $F$ is a list of $n$ elements that match $P_1$ through $P_n$.
- $P$ is of the form $(P_1 \ldots P_k P_e \ellipsis P_{m+1} \ldots P_n)$, 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.

```scheme
(syntax (template)) syntax
```

**Note:** #\(\text{'}\)(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
A (template) is a pattern variable, an identifier that is not a pattern variable, a pattern datum, or one of the following.

\[
\begin{align*}
&\text{(template) \ldots} \\
&\text{(subtemplate) \ldots (template) }
\end{align*}
\]

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 (\ldots) 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 \(#( t_1 \ldots t_n )\) is a vector if any of \( t_1 , \ldots , t_n \) 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.

\[
\begin{align*}
&\text{(define-syntax or} \\
&\text{\quad (lambda (x)} \\
&\quad \text{(syntax-case x ()} \\
&\quad \quad ([ ] (syntax #f)] \\
&\quad \quad ([e] (syntax e)] \\
&\quad \quad ([ e_1 e_2 e_3 \ldots ] \\
&\quad \quad \quad \text{ #'(let ([t e1])} \\
&\quad \quad \quad \quad \quad \text{ (if t t (or e2 e3 \ldots))]))))
\end{align*}
\]

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.

\[
\begin{align*}
&\text{(define p (cons 4 5)} \\
&\text{\quad (define-syntax p.car} \\
&\quad \text{\quad (lambda (x)} \\
&\quad \quad \text{(syntax-case x (set!)} \\
&\quad \quad \quad ([ ] (set! (car p) 15) \quad \Rightarrow 15 \\
&\quad \quad \quad (set! p.car 15) \quad \Rightarrow \text{&syntax exception}
\end{align*}
\]

12.5. Identifier predicates

\[
\text{(identifier? obj) procedure}
\]

Returns #t if \( obj \) is an identifier, i.e., a syntax object rep-
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=? id1 id2) procedure
Id1 and id2 must be identifiers. The procedure bound-identifier=? returns #t 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 [12.1]).

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=? id1 id2) procedure
Id1 and id2 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 [12.1]) 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 ...)])

(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 [12.4]. 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 ()
      ...))

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 [12.4]. Rather than including else in the literals list as before, this version explicitly tests for else using free-identifier=?.
With either definition of \texttt{case}, \texttt{else} is not recognized as an auxiliary keyword if an enclosing lexical binding for \texttt{else} exists. For example,

\begin{verbatim}
(let ((else #f))
  (case 0 [else (write "oops")]))
\end{verbatim}

⇒ \texttt{&syntax exception}

since \texttt{else} is bound lexically and is therefore not the same \texttt{else} that appears in the definition of \texttt{case}.

\subsection*{12.6. Syntax-object and datum conversions}

\texttt{(syntax->datum syntax-object)} \hfill procedure

The procedure \texttt{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 \texttt{eq?}. Thus, a predicate \texttt{symbolic-identifier=?} might be defined as follows.

\begin{verbatim}
(define symbolic-identifier=?
  (lambda (x y)
    (eq? (syntax->datum x) (syntax->datum y))))
\end{verbatim}

\texttt{(datum->syntax template-id datum)} \hfill procedure

\texttt{Template-id} must be a template identifier and \texttt{datum} should be a datum value. The \texttt{datum->syntax} procedure returns a syntax object representation of \texttt{datum} that contains the same contextual information as \texttt{template-id}, with the effect that the syntax object behaves as if it were introduced into the code when \texttt{template-id} was introduced.

The \texttt{datum->syntax} procedure allows a transformer to "bend" lexical scoping rules by creating \texttt{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 \texttt{loop} expression that uses this controlled form of identifier capture to bind the variable \texttt{break} to an escape procedure within the loop body. (The derived \texttt{with-syntax} form is like \texttt{let} but binds pattern variables—see section \texttt{[12.8]})

\begin{verbatim}
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(k e ...) ([break (datum->syntax #'k 'break)])
        #'(call-with-current-continuation
          (lambda (break)
            (let f () e ... (f))))])))
\end{verbatim}

(\texttt{let ((n 3) (ls '()))
  (loop
    (if (= n 0) (break ls))
    (set! ls (cons 'a ls))
    (set! n (- n 1))))

⇒ (a a a)

Were \texttt{loop} to be defined as

\begin{verbatim}
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(e ...) #'(call-with-current-continuation
          (lambda (break)
            (let f () e ... (f))))])))
\end{verbatim}

the variable \texttt{break} would not be visible in \texttt{e} ....

The \texttt{datum} argument \texttt{datum} may also represent an arbitrary Scheme form, as demonstrated by the following definition of \texttt{include}.

\begin{verbatim}
(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)
        ([fn (syntax->datum #'filename)]
          (with-syntax ([`exp ...]
             (read-file fn #'k))
            #'(begin exp ...)))]))
\end{verbatim}

\texttt{(include "filename"}) expands into a \texttt{begin} expression containing the forms found in the file named by \texttt{"filename"}. For example, if the file \texttt{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))))))

12.7. 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 equivalent to the one using syntax-rules given in report appendix B is shown below.

(define-syntax letrec
  (lambda (x)
    (syntax-case x ()
      ([ (i e) ...] b1 b2 ...)
      ([() (let () b1 b2 ...))]))

This version uses generate-temporaries instead of recursively defined helper to generate the necessary temporaries.

12.8. 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))

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 (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 case
  (lambda (x)
    (syntax-case x ()
      [(id x (...) ...) #'(exp1 x (...) ...)]
      [(set! var val) #'exp2]
      [(k ...) ((let f ([c1 #'c1] [cmore #'(c2 ...)])
          (if (null? cmore)
            (syntax-case c1 (else)
              [[(begin e1 e2 ...)]
                #'(begin e1 e2 ...)]
              [[(k ...) e1 e2 ...]
                #'(begin e1 e2 ...)])
            (cdr cmore))))]))))

12. syntax-case 55

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 template ...) is equivalent to (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 9.18.

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 identifier-syntax
  (syntax-rules (set!)
    [[(begin e1 e2 ...)]
      #'(if (memv t '(k ...))
          (begin e1 e2 ...))]
    [[(k ...) e1 e2 ...]
      #'(if (memv t '(k ...))
          (begin e1 e2 ...)
          (f (car cmore) (cdr cmore))))])))

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 9.18).

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.

As noted in report section 3.3.5 #’(template) is equivalent to (quasisyntax (template)), #'(template) is equivalent to (unsyntax (template)), and #,@(template) is equivalent to (unsyntax-splicing (template)).

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 ()
     [(id x (...) ...) #'(exp1 x (...) ...)]
     [(set! var val) #'exp2]
     [(k ...) ((let f ([c1 #'c1] [cmore #'(c2 ...)])
         (if (null? cmore)
           (syntax-case c1 (else)
             [[(begin e1 e2 ...)]
               #'(begin e1 e2 ...)]
             [[(k ...) e1 e2 ...]
               #'(begin e1 e2 ...)])
           (cdr cmore))))]))

A more robust implementation would verify that the literals (literals) ... 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 9.20) may be defined in terms of syntax-case, syntax, and make-variable-transformer as follows.

(define-syntax identifier-syntax
  (syntax-rules (set!)
    [[(begin e1 e2 ...)]
      #'(if (memv t '(k ...))
          (begin e1 e2 ...))]
    [[(k ...) e1 e2 ...]
      #'(if (memv t '(k ...))
          (begin e1 e2 ...)
          (f (car cmore) (cdr cmore))))])))
12.9. 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 7) 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.

13. Hashtables

The (rnrs hashtables (6)) library provides a set of operations on hashtables. A hashtable 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 exact integer objects. It is the programmer’s responsibility to ensure that the hash function is compatible with the equivalence function, which is a procedure that accepts two keys and returns true if they are equivalent and #f otherwise. Standard hashtables for arbitrary objects based on the eq? and eqv? predicates (see report section 9.6) are provided. Also, hash functions for arbitrary objects, strings, and symbols are provided.

This section uses the hashtable parameter name for arguments that must be hashtables, and the key parameter name for arguments that must be hashtable keys.

13.1. Constructors

(make-eq-hashtable)  procedure
(make-eq-hashtable k)  procedure

Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys, and compares those keys with eq?. If an argument is given, the initial capacity of the hashtable is set to approximately k elements.

(make-eqv-hashtable)  procedure
(make-eqv-hashtable k)  procedure

Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys, and compares those keys with eqv?. If an argument is given, the initial capacity of the hashtable is set to approximately k elements.

(make-hashtable hash-function equiv)  procedure
(make-hashtable hash-function equiv k)  procedure

Hash-function and equiv must be procedures. Hash-function should accept a key as an argument and should return a non-negative exact integer object. Equiv should accept two keys as arguments and return a single value. Neither procedure should mutate the hashtable returned by make-hashtable. The make-hashtable procedure returns a newly allocated mutable hashtable 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 hashtable is set to approximately k elements.

Both hash-function and 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 hashtable. Furthermore, any pair of keys for which equiv returns true should be hashed to the same exact integer objects by hash-function.

Implementation responsibilities: The implementation must check the restrictions on hash-function and equiv to the extent performed by applying them as described.

Note: Hashtables 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 hashtable operation may call the hash function more than once.
13. Hashtables

13.2. Procedures

(hashtable? hashtable)  procedure
Returns #t if hashtable is a hashtable, #f otherwise.

(hashtable-size hashtable)  procedure
Returns the number of keys contained in hashtable as an exact integer object.

(hashtable-ref hashtable key default)  procedure
Returns the value in hashtable associated with key. If hashtable does not contain an association for key, default is returned.

(hashtable-set! hashtable key obj)  procedure
Changes hashtable to associate key with obj, adding a new association or replacing any existing association for key, and returns unspecified values.

(hashtable-delete! hashtable key)  procedure
Removes any association for key within hashtable and returns unspecified values.

(hashtable-contains? hashtable key)  procedure
Returns #t if hashtable contains an association for key, #f otherwise.

(hashtable-update! hashtable key proc default)  procedure
Proc should accept one argument, should return a single value, and should not mutate hashtable. The hashtable-update! procedure applies proc to the value in hashtable associated with key, or to default if hashtable does not contain an association for key. The hashtable is then changed to associate key with the value returned by proc.

The behavior of hashtable-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:

(hashtable-set!
 hashtable key
 (proc (hashtable-ref
          hashtable key default)))

(hashtable-copy hashtable)  procedure
Returns a copy of hashtable. If the mutable argument is provided and is true, the returned hashtable is mutable; otherwise it is immutable.

(hashtable-clear! hashtable)  procedure
Returns a copy of hashtable. The order of the vector is unspecified.

(hashtable-equivalence-function hashtable)  procedure
Returns the equivalence function used by hashtable to compare keys. For hashtables created with make-eq-hashtable and make-eqv-hashtable, returns eq? and eqv? respectively.

(hashtable-hash-function hashtable)  procedure
Returns the hash function used by hashtable. For hashtables created by make-eq-hashtable or make-eqv-hashtable, #f is returned.

(hashtable-mutable? hashtable)  procedure
Returns #t if hashtable is mutable, otherwise #f.

13.3. Inspection

(hashtable-entries hashtable)  procedure
Returns two values, a vector of the keys in hashtable, and a vector of the corresponding values.

(let ((h (make-eqv-hashtable)))
  (hashtable-set! h 1 'one)
  (hashtable-set! h 2 'two)
  (hashtable-set! h 3 'three)
  (hashtable-entries h))
⇒ (#(1 2 3), #(one two three))
; two return values
13.4. Hash functions

The equal-hash, string-hash, and string-ci-hash procedures of this section are acceptable as the hash functions of a hashtable only if the keys on which they are called are not mutated while they remain in use as keys in the hashtable.

(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.

Note: Like equal?, the equal-hash procedure must always terminate, even if its arguments contain cycles.

(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.

14. Enumerations

This chapter describes the (rnrs enum (6)) 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 descriptions of the following procedures, enum-set ranges over the enumeration sets, which are defined as the subsets of the universes that can be defined using make-enumeration.

(make-enumeration symbol-list) procedure

Symbol-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, as an enumeration set.

(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 (rnrs lists (6)) 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. The values in the list must all belong to the universe.

(enum-set->list 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

Returns #t if symbol is in the universe of enum-set. Otherwise, #f.

(ensemble-subset? ensemble1 ensemble2) procedure

Returns #t if ensemble1 is a subset of ensemble2. Otherwise, #f.

(ensemble=? ensemble1 ensemble2) procedure

Returns #t if the universes of ensemble1 and ensemble2 are equal. Otherwise, #f.

The enum-set-member? procedure returns #t if its first argument is an element of its second argument, #f otherwise.

The ensemble-subset? procedure returns #t if the universe of ensemble1 is a subset of the universe of ensemble2 (considered as sets of symbols) and every element of ensemble1 is a member of ensemble2. It returns #f otherwise.

The ensemble=? procedure returns #t if 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 #t #t #t #f #t #f #t)

(ensemble-union ensemble1 ensemble2) procedure

Projects ensemble1 into the universe of ensemble2, dropping any elements of ensemble1 that do not belong to the universe of ensemble2. (If ensemble1 is a subset of the universe of its second, no elements are dropped, and the injection is returned.)

(let ((e1 (make-enumeration '(red green blue)))
      (e2 (make-enumeration '(red white)))
      (e (make-enumeration '(red blue)))))
  (ensemble-union e1 e2)
⇒ (red blue)

(ensemble-intersection ensemble1 ensemble2) procedure

Projects ensemble1 into the universe of ensemble2, dropping any elements of ensemble1 that do not belong to the universe of ensemble2. (If ensemble1 is a subset of the universe of its second, no elements are dropped, and the injection is returned.)

(let ((e1 (make-enumeration '(red green blue)))
      (e2 (make-enumeration '(red white)))
      (e (make-enumeration '(red blue))))
  (ensemble-intersection e1 e2))
⇒ (red black)

(ensemble-difference ensemble1 ensemble2) procedure

Projects ensemble1 into the universe of ensemble2, dropping any elements of ensemble1 that do not belong to the universe of ensemble2. (If ensemble1 is a subset of the universe of its second, no elements are dropped, and the injection is returned.)

(let ((e1 (make-enumeration '(red green blue)))
      (e2 (make-enumeration '(red white)))
      (e (make-enumeration '(red blue))))
  (ensemble-difference e1 e2))
⇒ (red black)

(ensemble-complement ensemble) procedure

Projects ensemble into its universe, possibly with duplicates, expands into an expression that evaluates to the enumeration set of those symbols.

(let ((e (make-enumeration '(red green blue))))
  (ensemble-complement e))
⇒ (green blue)

(ensemble-projection ensemble1 ensemble2) procedure

Projects ensemble1 into the universe of ensemble2, dropping any elements of ensemble1 that do not belong to the universe of ensemble2. (If ensemble1 is a subset of the universe of its second, no elements are dropped, and the injection is returned.)

(let ((e1 (make-enumeration '(red green blue)))
      (e2 (make-enumeration '(red white))))
  (ensemble-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) (symbol) ...⟩ checks at macro-expansion time whether ⟨symbol⟩ is in the universe associated with ⟨type-name⟩. If it is, ⟨(type-name) ⟨symbol⟩⟩ is equivalent to ⟨symbol⟩. It is a syntax violation if it is not.

⟨(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

\[
\text{((enum-set-constructor ((constructor-syntax)) '((symbol) ...)))}
\]

Example:

```
(define-enumeration color
  (black white purple maroon)
  color-set)

(color black)  ⇒  black
(color purple)  ⇒  &syntax exception
(enum-set->list (color-set))  ⇒  ()
(enum-set->list
  (color-set maroon white))  ⇒  (white maroon)
```

15. Composite library

The `(rnrs (6))` library is a composite of most of the libraries described in this report. The only exceptions are:

- `(rnrs eval (6))` (chapter 16)
- `(rnrs mutable-pairs (6))` (chapter 17)
- `(rnrs mutable-strings (6))` (chapter 18)
- `(rnrs r5rs (6))` (chapter 19)

The library exports all procedures and syntactic forms provided by the component libraries.

All of the bindings exported by `(rnrs (6))` are exported for both `run` and `expand`; see report section 6.2.

16. eval

The `(rnrs eval (6))` 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 determined not to be 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 is determined to contain an assignment to one of the variables of the environment, then `eval` must raise an exception with a condition type `&assertion`.

```
(library (foo)
  (export)
  (import (rnrs (6)))
  (write
    (eval '(let ((x 3)) x)
      (environment '((rnrs (6)))))
  )
writes 3

(library (foo)
  (export)
  (import (rnrs (6)))
  (write
    (eval
      '(eval:car (eval:cons 2 4))
      (environment
        '(prefix (only (rnrs (6)) car cdr cons null?)
          eval:))))
  )
writes 2
```

17. Mutable pairs

The procedures provided by the `(rnrs mutable-pairs (6))` library allow new values to be assigned to the `car` and `cdr` fields of previously allocated pairs.

```
(set-car! pair obj)  procedure
```

Stores `obj` in the car field of `pair`. The `set-car!` procedure returns unspecified values.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)  ⇒  unspecified
(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`. The `set-cdr!` procedure returns unspecified values.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-cdr! (f) 3)  ⇒  unspecified
(set-cdr! (g) 3)  ⇒  unspecified
; should raise &assertion exception
```

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? y y)
     (equal? (list x y 'a) (list y x 'b)))
⇒ (#t #t #f)

18. Mutable strings
The string-set! procedure provided by the (rnrs mutable-strings (6)) library allows mutating the characters of a string in-place.

(string-set! string k char) procedure
K must be a valid index of string. The string-set! procedure stores char in element k of string and returns unspecified values.

Passing an immutable string to string-set! should cause an exception with condition type &assertion to be raised.

(define (f) (make-string 3 '#*'))
(define (g) "***")
(string-set! (f) 0 '#?) ⇒ unspecified
(string-set! (g) 0 '#?) ⇒ unspecified
    ; should raise &assertion exception
(string-set! (symbol->string 'immutable)
    0
    '#?) ⇒ unspecified
    ; should raise &assertion exception

Note: Implementors are encouraged to make string-set! run in constant time.

(string-fill! string char) procedure
Stores char in every element of the given string and returns unspecified values.

19. R5RS compatibility
The features described in this chapter are exported from the (rnrs r5rs (6)) library and provide some functionality of the preceding revision of this report [6] that was omitted from the main part of the current report.

(quotient n1 n2) procedure
(remainder n1 n2) procedure
(modulo n1 n2) procedure

These procedures implement number-theoretic (integer) division. N2 must be non-zero. All three procedures return integer objects. If n1/n2 is an integer object:

(quotient n1 n2) ⇒ n1/n2
(remainder n1 n2) ⇒ 0
(modulo n1 n2) ⇒ 0

If n1/n2 is not an integer object:

(quotient n1 n2) ⇒ nq
(remainder n1 n2) ⇒ nr
(modulo n1 n2) ⇒ nm

where nq is n1/n2 rounded towards zero, 0 < |nr| < |n2|, 0 < |nm| < |n2|, nr and nm differ from n1 by a multiple of n2, nr has the same sign as n1, and nm has the same sign as n2.

Consequently, for integer objects n1 and n2 with n2 not equal to 0,

(= n1 (+ (* n2 (quotient n1 n2)))
  (remainder n1 n2))) ⇒ #t

provided all number object 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
(modulo -13 -4.0) ⇒ -1.0 ; inexact

Note: These procedures could be defined in terms of div and mod (see report section 9.8.4) 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
(p begin (set! x 10))
(continue 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)))))))

(null-environment n)  procedure

N must be the exact integer object 5. The null-environment procedure returns an environment specifier suitable for use with eval (see chapter 10) representing an environment that is empty except for the (syntactic) bindings for all keywords described in the previous revision of this report [6].

(scheme-report-environment n)  procedure

N must be the exact integer object 5. The scheme-report-environment procedure returns an environment specifier for an environment that is empty except for the bindings for the identifiers described in the previous revision of this report [6], omitting load, transcript-on, transcript-off, and char-ready?. The bindings have as values the procedures of the same names described in this report.

REFERENCES


ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

antimark 48
&assertion 27
assertion-violation? 27
assoc 12
assp 12
assq 12
assv 12

base record type 16
big-endian 5
binary port 29, 30
binary-port? 32
bit fields 46
bitwise-and 46
bitwise-arithmetic-shift 47
bitwise-arithmetic-shift-left 47
bitwise-arithmetic-shift-right 47
bitwise-bit-count 46
bitwise-bit-field 47
bitwise-bit-set? 47
bitwise-copy-bit 47
bitwise-copy-bit-field 47
bitwise-first-bit-set 47
bitwise-if 46
bitwise-ior 46
bitwise-length 46
bitwise-not 46
bitwise-reverse-bit-field 47
bitwise-rotate-bit-field 47
bitwise-xor 46
bound-identifier=? 52
buffer-mode 30
buffer-mode? 30
byte 5
bytevector 5
bytevector->sint-list 7
bytevector->string 32
bytevector->u8-list 6
bytevector->uint-list 7
bytevector-copy 6
bytevector-copy! 6
bytevector-fill! 6
bytevector-ieee-double-native-ref 9
bytevector-ieee-double-native-set! 9
bytevector-ieee-double-ref 9
bytevector-ieee-single-native-ref 9
bytevector-ieee-single-native-set! 9
bytevector-ieee-single-ref 9
bytevector-length 9
bytevector-s16-native-ref 7
bytevector-s16-native-set! 7
bytevector-s16-ref 7
bytevector-s16-set! 7
bytevector-s32-native-ref 8
bytevector-s32-native-set! 8
bytevector-s32-ref 8
bytevector-s32-set! 8
bytevector-s64-native-ref 8
bytevector-s64-native-set! 8
bytevector-s64-ref 8
bytevector-s64-set! 8
bytevector-s8-ref 9
bytevector-s8-set! 9
bytevector-sint-ref 6
bytevector-sint-set! 6
bytevector-string 32
bytevector-uint-ref 6
bytevector-uint-set! 6
bytevector=? 6
bytevector? 6

call by need 62
call-with-bytevector-output-port 37
call-with-input-file 40
call-with-output-file 40
call-with-port 39
call-with-string-output-port 38
case-lambda 14, 15
char-alphabetic? 3
char-ci<=? 3
char-ci<=? 3
char-ci>? 3
char-ci>? 3
char-downcase 3
char-foldcase 3
char-general-category 3
char-lower-case? 3
char-numeric? 3
flan? 44
finlinear? 44
flnumerator 45
flodd? 44
flonum? 44
flpositive? 44
flround 45
flsin 45
flsqrt 46
fltan 45
fltruncate 45
flush-output-port 37
flzero? 44
fold-left 11
fold-right 11
for-all 10
force 62
free-identifier=? 42
fx* 42
fx*/carry 12
fx+ 42
fx+/carry 12
fx- 12
fx-/>carry 12
fx⇐? 11
fx?= 11
fx=>? 11
fx?> 11
fx> 11
fxand 42
fxarithmetic-shift 43
fxarithmetic-shift-left 13
fxarithmetic-shift-right 43
fxbits-count 43
fxbits-field 43
fxbits-set? 43
fxcopy-bit 43
fxcopy-bit-field 43
fxdiv 42
fxdiv-and-mod 42
fxdiv0 12
fxdiv0-and-mod0 42
fxeven? 42
fxfirst-bit-set 43
fxif 42
fxior 42
fxlength 43
fxmax 42
fxmin 42
fxmod 42
fxmod0 42
fxnegative? 42
fxnot 42
fxodd? 42
fxpositive? 42
fxreverse-bit-field 44
fxrotate-bit-field 43
fxxor 12
fxzero? 12
generate-temporaries 54
get-bytevector-all 35
get-bytevector-n 35
get-bytevector-n! 35
get-bytevector-some 35
get-char 36
get-datum 36
get-line 36
get-string-all 36
get-string-n 36
get-string-n! 36
get-u8 35
guard 23
hash function 56
hashtable 56
hashtable-clear! 57
hashtable-contains? 57
hashtable-copy 57
hashtable-delete! 57
hashtable-entries 57
hashtable-equivalence-function 57
hashtable-hash-function 57
hashtable-keys 57
hashtable-mutable? 57
hashtable-ref 57
hashtable-set! 57
hashtable-size 57
hashtable-update! 57
hashtable? 57
&i/o 27
&i/o-decoding 31
i/o-decoding-error-transcoder 31
i/o-decoding-error? 31
&i/o-encoding 31
i/o-encoding-error-char 31
i/o-encoding-error-transcoder 31
i/o-encoding-error? 31
i/o-error-filename 28
i/o-error-port 29
i/o-error? 27
&i/o-file-already-exists 28
i/o-file-already-exists-error? 28
&i/o-file-does-not-exist 28
i/o-file-does-not-exist-error? 28
&i/o-file-is-read-only 28
i/o-file-is-read-only-error? 28
&i/o-file-protection 28
i/o-file-protection-error? 28
&i/o-filename 28
i/o-filename-error? 28
&i/o-invalid-position 28
i/o-invalid-position-error? 28
&i/o-port 29
i/o-port-error? 29
&i/o-read 28
i/o-read-error? 28
&i/o-write 28
i/o-write-error? 28
identifier 49
identifier macro 51
identifier? 51
immutable record type 16
&implementation-restriction 26
implementation-restriction-violation? 26
implicit identifier 53
inexact->exact 61
input port 29
input-port? 33
&irritants 27
irritants-condition? 27
latin-1-codec 30
lazy evaluation 62
&lexical 26
lexical-violation? 26
library specifier 60
list-sort 13
little-endian 5
lookahead-char 36
lookahead-u8 35
macro transformer 49
make-assertion-violation 27
make-bytevector 5
make-custom-binary-input-port 34
make-custom-binary-input/output-port 39
make-custom-binary-output-port 35
make-custom-textual-input-port 34
make-custom-textual-input/output-port 39
make-custom-textual-output-port 38
make-enumeration 58
make-eq-hashtable 56
make-eqv-hashtable 54
make-error 26
make-hashable 58
make-i/o-decoding-error 31
make-i/o-encoding-error 31
make-i/o-error 27
make-i/o-file-already-exists-error 28
make-i/o-file-does-not-exist-error 28
make-i/o-file-is-read-only-error 28
make-i/o-file-protection-error 28
make-i/o-filename-error 28
make-i/o-invalid-position-error 28
make-i/o-port-error 29
make-i/o-read-error 28
make-i/o-write-error 28
make-implementation-restriction-violation 26
make-irritants-condition 27
make-lexical-violation 26
make-message-condition 26
make-no-infinities-violation 46
make-no-nans-violation 46
make-non-continuable-violation 26
make-record-constructor-descriptor 17
make-record-type-descriptor 16
make-serious-condition 26
make-syntax-violation 27
make-transcoder 32
make-undefined-violation 27
make-variable-transformer 49
make-violation 26
make-warning 26
make-who-condition 27
mark 48
member 12
memp 12
memq 12
memv 12
&message 26
message-condition? 26
modulo 61
mutable record type 16
native-endianness 6
native-eol-style 30
native-transcoder 32
newline 10
&no-infinities 46
no-infinities-violation? 46
&no-nans 46
no-nans-violation? 46
&non-continuable 26
non-continuable-violation? 26
null-environment 62
number 41
octet 5
open-bytevector-input-port 33
open-bytevector-output-port 37
open-file-input-port 33
open-file-input/output-port 39
open-file-output-port 37
open-input-file 40
open-output-file 40
open-string-input-port 34
open-string-output-port 37
output ports 29
output-port-buffer-mode 37
output-port? 37
partition
pattern variable
peek-char
port
port-eof?
port-has-port-position?
port-has-set-port-position?
port-position
port-transcoder
port?
position
promise
protocol
put-bytevector
put-char
put-datum
put-string
put-u8
quasixyntax
quotient
raise
raise-continuable
read
read-char
real->flonum
record
record constructor
record-accessor
record-constructor
record-constructor descriptor
record-field-mutable?
record-mutator
record-predicate
record-rtd
record-type descriptor
record-type-descriptor
record-type-field-names
record-type-generative?
record-type-name
record-type-opaque?
record-type-parent
record-type-sealed?
record-type-uid
record?
region
remainder
remove
remq
remv
(rnrs (6))
(rnrs arithmetic bitwise (6))
(rnrs arithmetic flonum (6))
(rnrs arithmetic fx (6))
(rnrs bytevector (6))
(rnrs conditions (6))
(rnrs control (6))
(rnrs enum (6))
(rnrs exceptions (6))
(rnrs files (6))
(rnrs hashtables (6))
(rnrs io ports (6))
(rnrs io simple (6))
(rnrs lists (6))
(rnrs mutable-pairs (6))
(rnrs mutable-strings (6))
(rnrs programs (6))
(rnrs r5rs (6))
(rnrs records inspection (6))
(rnrs records procedural (6))
(rnrs records syntactic (6))
(rnrs sorting (6))
(rnrs syntax-case (6))
(rnrs unicode (6))
scheme-report-environment
&serious
serious-condition?
set-car!
set-cdr!
set-port-position!
simple condition
simple-conditions
sint-list->bytevector
standard-error-port
standard-input-port
standard-output-port
string->bytevector
string->utf16
string->utf32
string->utf8
string-ci-hash
string-ci<=?
string-ci?>
string-ci<=?
string-ci=>?
string-ci>=?
string-downcase
string-fill!
string-foldcase
string-hash
string-normalize-nfc
string-normalize-nfd
string-normalize-nfkc
string-normalize-nfkf
string-set!
Index 69

string-titlecase 4
string-upcase 4
substitution 48
symbol-hash 58
&syntax 27
syntax 50
syntax object 49
syntax->datum 53
syntax-case 49
syntax-violation 56
syntax-violation-form 27
syntax-violation-subform 27
syntax-violation? 27
textual port 30
textual ports 29
textual-port? 32
transcoded-port 32
transcoder 30
transcoder-codec 32
transcoder-eol-style 32
transcoder-error-handling-mode 32
transformation procedure 49
transformer 49
u8-list->bytevector 6
uint-list->bytevector 7
&undefined 27
undefined-violation? 27
universe 58
unless 13, 14
utf-16-codec 30
utf-8-codec 30
utf16->string 10
utf32->string 10
utf8->string 10
variable transformer 49
vector-sort 13
vector-sort! 13
&violation 26
violation? 26
&warning 26
warning? 26
when 13, 14
&who 27
who-condition? 27
with-exception-handler 23
with-input-from-file 40
with-output-to-file 40
with-syntax 54
wrap 48
wrapped syntax object 48
write 41
write-char 40