Revised\textsuperscript{6} Report on the Algorithmic Language Scheme

— Standard Libraries —

MICHAEL SPERBER
R. KENT DYBVIG, MATTHEW FLATT, ANTON VAN STRAATEN
(Editors)

RICHARD KELSEY, WILLIAM CLINGER, JONATHAN REES
(Editors, Revised\textsuperscript{5} Report on the Algorithmic Language Scheme)

26 September 2007

**Unofficial version incorporating errata**

https://standards.scheme.org/

21 July 2019

SUMMARY

The report gives a defining description of the standard libraries of the programming language Scheme.

This report frequently refers back to the Revised\textsuperscript{6} Report on the Algorithmic Language Scheme [1]: 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 [3]. We gratefully acknowledge their authors for their contributions. More detailed information on authorship can be found at the beginning of the Revised\textsuperscript{6} 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.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<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>14</td>
</tr>
<tr>
<td>6.1</td>
<td>Mutability and equivalence of records</td>
<td>16</td>
</tr>
<tr>
<td>6.2</td>
<td>Syntactic layer</td>
<td>16</td>
</tr>
<tr>
<td>6.3</td>
<td>Procedural layer</td>
<td>20</td>
</tr>
<tr>
<td>6.4</td>
<td>Inspection</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Exceptions and conditions</td>
<td>24</td>
</tr>
<tr>
<td>7.1</td>
<td>Exceptions</td>
<td>24</td>
</tr>
<tr>
<td>7.2</td>
<td>Conditions</td>
<td>26</td>
</tr>
<tr>
<td>7.3</td>
<td>Standard condition types</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>I/O</td>
<td>29</td>
</tr>
<tr>
<td>8.1</td>
<td>Condition types</td>
<td>30</td>
</tr>
<tr>
<td>8.2</td>
<td>Port I/O</td>
<td>31</td>
</tr>
<tr>
<td>8.3</td>
<td>Simple I/O</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>File system</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Command-line access and exit values</td>
<td>43</td>
</tr>
<tr>
<td>11</td>
<td>Arithmetic</td>
<td>43</td>
</tr>
<tr>
<td>11.1</td>
<td>Bitwise operations</td>
<td>43</td>
</tr>
<tr>
<td>11.2</td>
<td>Fixnums</td>
<td>43</td>
</tr>
<tr>
<td>11.3</td>
<td>Flonums</td>
<td>46</td>
</tr>
<tr>
<td>11.4</td>
<td>Exact bitwise arithmetic</td>
<td>49</td>
</tr>
<tr>
<td>12</td>
<td><code>syntax-case</code></td>
<td>50</td>
</tr>
<tr>
<td>12.1</td>
<td>Hygiene</td>
<td>50</td>
</tr>
<tr>
<td>12.2</td>
<td>Syntax objects</td>
<td>51</td>
</tr>
<tr>
<td>12.3</td>
<td>Transformers</td>
<td>52</td>
</tr>
<tr>
<td>12.4</td>
<td>Parsing input and producing output</td>
<td>52</td>
</tr>
<tr>
<td>12.5</td>
<td>Identifier predicates</td>
<td>54</td>
</tr>
<tr>
<td>12.6</td>
<td>Syntax-object and datum conversions</td>
<td>56</td>
</tr>
<tr>
<td>12.7</td>
<td>Generating lists of temporaries</td>
<td>57</td>
</tr>
<tr>
<td>12.8</td>
<td>Derived forms and procedures</td>
<td>57</td>
</tr>
<tr>
<td>12.9</td>
<td>Syntax violations</td>
<td>58</td>
</tr>
<tr>
<td>13</td>
<td>Hashtables</td>
<td>59</td>
</tr>
<tr>
<td>13.1</td>
<td>Constructors</td>
<td>59</td>
</tr>
<tr>
<td>13.2</td>
<td>Procedures</td>
<td>59</td>
</tr>
<tr>
<td>13.3</td>
<td>Inspection</td>
<td>60</td>
</tr>
<tr>
<td>13.4</td>
<td>Hash functions</td>
<td>60</td>
</tr>
<tr>
<td>14</td>
<td>Enumerations</td>
<td>60</td>
</tr>
<tr>
<td>15</td>
<td>Composite library</td>
<td>62</td>
</tr>
<tr>
<td>16</td>
<td><code>eval</code></td>
<td>63</td>
</tr>
<tr>
<td>17</td>
<td>Mutable pairs</td>
<td>63</td>
</tr>
<tr>
<td>18</td>
<td>Mutable strings</td>
<td>63</td>
</tr>
<tr>
<td>19</td>
<td>R5RS compatibility</td>
<td>64</td>
</tr>
<tr>
<td>References</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Alphabetic index of definitions of concepts, keywords, and procedures</td>
<td>67</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 [12].

Some of the procedures that operate on characters or strings ignore the difference between upper case and lower case. These procedures 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 case-folded character, then char-foldcase returns that character. Otherwise the character returned is the same as the argument. For Turkish characters Ł (#\x130) and Ł (#\x131), char-foldcase behaves as the identity function; otherwise char-foldcase is the same as char-downcase composed with char-upcase.

(char-upcase #\i) ⇒ #\i
(char-downcase #\i) ⇒ #\i
(char-titlecase #\i) ⇒ #\i
(char-foldcase #\i) ⇒ #\i

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

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

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

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 Turkish mappings in the latter.

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

(char-ci=? char1 char2 char3 ...) procedure
(char-ci<? char1 char2 char3 ...) procedure
(char-ci>? char1 char2 char3 ...) procedure
(char-ci<=? char1 char2 char3 ...) procedure
(char-ci>=? char1 char2 char3 ...) procedure

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

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

(char-alphabetic? char) procedure
(char-numeric? char) procedure
(char-whitespace? char) procedure
(char-upper-case? 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 it 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.
Returns a symbol representing the Unicode general category of char, one of Lu, Li, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Ps, Pe, Pi, Pf, Pd, Pc, Po, Sc, Sm, Sk, So, Zs, Zp, Zl, Cc, Cf, Cs, Co, or Cn.

(char-general-category #\a)  \(\rightarrow\) Ll
(char-general-category #\space)  \(\rightarrow\) Za
(char-general-category #\x10FFFF)  \(\rightarrow\) Lm

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, and downcases all other cased characters.

(string-upcase "Hi")  \(\rightarrow\) "HI"
(string-downcase "Hi")  \(\rightarrow\) "hi"
(string-foldcase "Hi")  \(\rightarrow\) "hi"
(string-upcase "Straße")  \(\rightarrow\) "STRASSE"
(string-downcase "Straße")  \(\rightarrow\) "strasse"
(string-foldcase "Straße")  \(\rightarrow\) "strasse"
(string-downcase "Σ")  \(\rightarrow\) "σ"

(string-titlecase "ΧΑΟΣ" )  \(\rightarrow\) "ΧΑΟΣ"
(string-downcase "ΧΑΟΣ")  \(\rightarrow\) "χαος"

(string-titlecase "ＫnOck KNoCK")  \(\rightarrow\) "Knock Knock"
(string-titlecase "who's there?")  \(\rightarrow\) "Who's There?"
(string-titlecase "r6rs")  \(\rightarrow\) "R6rs"
(string-titlecase "r6rs")  \(\rightarrow\) "R6rs"

Note: The case mappings needed for implementing these procedures can be extracted from UnicodeData.txt, SpecialCasing.txt, WordBreakProperty.txt, 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].
2. Bytevectors

Many applications deal with blocks of binary data by accessing them in various ways—extracting signed or unsigned numbers of various sizes. Therefore, the (rnrs bytevectors (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.

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 less than the length of the bytevector. 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 string literals, literals representing bytevectors do not need to be quoted:

\[
\text{#vu8}(12 \ 23 \ 123) \quad \Rightarrow \ #vu8(12 \ 23 \ 123)
\]

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

\begin{align*}
\text{(endianness } & \text{endianness symbol)} \quad \text{syntax} \\
\text{(native-endianness)} & \quad \text{procedure} \\
\text{(bytevector? obj)} & \quad \text{procedure} \\
\text{(make-bytevector k)} & \quad \text{procedure} \\
\text{(make-bytevector k fill)} & \quad \text{procedure} \\
\text{(bytevector-length bytevector)} & \quad \text{procedure}
\end{align*}

The name of \text{endianness symbol} 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})} evaluates to the symbol named \text{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 \text{endianness symbol} to be anything other than an endianness symbol supported by the implementation.

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

Note: Only the name of \text{endianness symbol} is significant.

\text{(bytevector? obj)}

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

\text{(make-bytevector k)}

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)}

Returns, as an exact integer object, the number of bytes in \text{bytevector}.
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.

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

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.

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.

(2) 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.

The bytevector-copy! procedure returns a newly allocated copy of bytevector.

2.3. Operations on bytes and octets

(3) 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.

2.4. Operations on integers of arbitrary size
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}`.

The `bytevector-sint-set!` procedure 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)
⇒ #xFFFFFFFFFFFFFFFFFFFFFFFF

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

(bytevector-uint-set! (u8-list->bytevector '(1 2 3 255 1 2 1 2)) (endianness little) 2)
⇒ (513 -253 513 513)

(let ((b (u8-list->bytevector '(1 2 3 255 1 2 1 2))))
  (bytevector-uint-ref b 0 (endianness little) 2))
⇒ (513 65283 513 513)
```

2.5. Operations on 16-bit integers

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

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

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

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

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

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

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

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

2.6. Operations on 32-bit integers

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

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

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

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

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

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

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

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

```
K, . . . , k + 3 must be valid indices of bytevector.
```

```
K, . . . , k + 7 must be valid indices of bytevector.
```

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

The . . . procedures return unspecified values.

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

```
(bytevector-u32-ref b 12 (endianness little))
⇒ 4261412863
```

```
(bytevector-s32-ref b 12 (endianness little))
⇒ -33554433
```

```
(bytevector-u32-ref b 12 (endianness big))
⇒ 4294967293
```

```
(bytevector-s32-ref b 12 (endianness big))
⇒ -3
```

2.7. Operations on 64-bit integers

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

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

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

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

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

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

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

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

```
K, . . . , k + 7 must be valid indices of bytevector.
```

```
K, . . . , k + 7 must be valid indices of bytevector.
```

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

The . . . procedures return unspecified values.

```
(define b (u8-list->bytevector
```

2.8. Operations on IEEE-754 representations

(bytevector-ieee-single-native-ref bytevector k)  procedure
K, ..., k + 3 must be valid indices of bytevector. For
(bytevector-ieee-single-ref bytevector k endianness)  procedure
(bytevector-ieee-double-native-ref bytevector k)  procedure
K, ..., k + 7 must be valid indices of bytevector. For
(bytevector-ieee-double-ref bytevector k endianness)  procedure
These procedures return the inexact real number object
(bytevector-ieee-single-native-set! bytevector k x)  procedure
that best represents the IEEE-754 single-precision number
K, ..., k + 3 must be valid indices of bytevector. For
represented by the four bytes beginning at index k.
(bytevector-ieee-single-set! bytevector k x endianness)  procedure
These procedures return the inexact real number object
(bytevector-ieee-double-native-set! bytevector k x)  procedure
that best represents the IEEE-754 double-precision number
K, ..., k + 7 must be valid indices of bytevector. For
represented by the eight bytes beginning at index k.
(bytevector-ieee-double-native-set! bytevector k x endianness)  procedure
These procedures store an IEEE-754 single-precision repre-
These procedures store an IEEE-754 double-precision rep-
sentation of x into elements k through k + 7 of bytevector,
tentation of x into elements k through k + 7 of bytevector,
and return unspecified values.
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 er-
ors are handled as with the replace semantics of textual
I/O (see section 8.2.4): If an invalid or incomplete char-
acter encoding is encountered, then the replacement char-
acter U+FFFD is appended to the string being generated,
an appropriate number of bytes are ignored, and decoding
continues with the following bytes.

(string->utf8 string)  procedure
Returns a newly allocated (unless empty) bytevector that
contains the UTF-8 encoding of the given string.

(string->utf16 string)  procedure
(string->utf16 string endianness)  procedure
If endianness is specified, it must be the symbol big or the
symbol little. The string->utf16 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.

(string->utf32 string)  procedure
(string->utf32 string endianness)  procedure
If endianness is specified, it must be the symbol big or the
symbol little. The string->utf32 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?)

Endianness 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. Bytvector is decoded according to UTF-16, UTF-16BE, UTF-16LE, or a fourth encoding scheme that differs from all three of those as follows: If endianness-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-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 #x00, #xFF specifying big and UTF-16BE, or #xFF, #xFE specifying little and UTF-16LE.

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

Endianness 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. Bytvector is decoded according to UTF-32, UTF-32BE, UTF-32LE, or a fourth encoding scheme that differs from all three of those as follows: If endianness-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-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 #x00, #xFF, #xFE, specifing big and UTF-32BE, or #x00, #xFF, #xFE specifying little and UTF-32LE.

3. List utilities
This chapter describes the (rnrs lists (6)) library, which contains various useful procedures that operate on lists.

(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 5 1 5 9)) ⇒ #f

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

(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 x_1^n, where x_i is the ith element of list_i, until #f is returned. If proc returns true values for all but the last element of list_1, for-all performs a tail call of proc on the kth elements, where k is the length of list_1. If proc returns #f on any set of elements, for-all returns #f after the first such application of proc. If the lists are all empty, for-all returns #t.

For natural numbers i = 0, 1, ..., the exists procedure applies proc successively to arguments x_1^n, where x_i is the ith element of list_i, 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 list_1. 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? '(2 4 14)) ⇒ #t
(for-all even? '(2 4 14 . 9)) ⇒ #assertion exception
(for-all (lambda (n) (and (even? n) n)) ' (2 4 14)) ⇒ 14
(for-all < '(1 2 3) '(2 3 4)) ⇒ #t
(for-all < '(1 2 4) '(2 3 4)) ⇒ #f
Implementation responsibilities: The implementation must check that the lists are chains of pairs to the extent necessary to determine the return value. If this requires traversing the lists entirely, the implementation should check that the lists all have the same length. If not, it should not check that the lists are chains beyond the traversal. The implementation must check the restrictions on proc to the extent performed by applying it as described. An implementation may check whether proc is an appropriate argument before applying it.

\[
\begin{align*}
&\text{(exists even? '(3 1 4 1 5 9))} \quad \Rightarrow \#t \\
&\text{(exists even? '(3 1 1 5 9))} \quad \Rightarrow \#f \\
&\text{(exists even? '(3 1 1 5 9 . 2))} \quad \Rightarrow \text{assertion exception} \\
&\text{(exists (lambda (n) (and (even? n) n)) '(2 1 4 14))} \quad \Rightarrow 2 \\
&\text{(exists < '(1 2 4) '(2 3 4))} \quad \Rightarrow \#t \\
&\text{(exists > '(1 2 3) '(2 3 4))} \quad \Rightarrow \#f \\
\end{align*}
\]

Proc should accept one argument and return a single value. Proc should not mutate list.

The 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 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 filter or, respectively, partition itself. If multiple returns occur from filter or partitions, the return values returned by earlier returns are not mutated.

\[
\begin{align*}
&\text{(filter even? '(3 1 4 1 5 9 2 6))} \quad \Rightarrow (4 2 6) \\
&\text{(partition even? '(3 1 4 1 5 9 2 6))} \quad \Rightarrow (4 2 6) (3 1 1 5 9) \text{; two values}
\end{align*}
\]

Implementation responsibilities: The implementation must check the restrictions on proc to the extent performed by applying it as described. An implementation may check whether proc is an appropriate argument before applying it.

\[
\begin{align*}
&\text{(fold-left combine nil list \ldots list)} \quad \text{procedure} \\
&\text{(fold-right combine nil list \ldots list)} \quad \text{procedure}
\end{align*}
\]

The lists should all have the same length. 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 fold-left procedure iterates the combine procedure over an accumulator value and the elements of the lists from left to right, starting with an accumulator value of nil. More specifically, fold-left returns nil if the lists are empty. If they are not empty, combine is first applied to 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. Combine is always called in the same dynamic environment as fold-left itself.

\[
\begin{align*}
&\text{(fold-left + 0 '(1 2 3 4 5))} \Rightarrow 15 \\
&\text{(fold-left (lambda (a e) (cons e a)) '(())} \quad \Rightarrow (5 4 3 2 1) \\
&\text{(fold-left (lambda (count x) (if (odd? x) (+ count 1) count))} \\
&\text{0} \quad \Rightarrow (7) \\
&\text{(fold-left (lambda (max-len s) (max max-len (string-length s)))} \\
&\text{0} \quad \Rightarrow (7) \\
&\text{(fold-left cons '(q) '(a b c))} \quad \Rightarrow (((q) . a) . b) . c) \\
&\text{(fold-left + 0 '(1 2 3) '(4 5 6))} \quad \Rightarrow 21
\end{align*}
\]
elements of the *lists* in order and *nil*. The result becomes the new accumulator value, and `combine` is applied to the respective previous elements of the *lists* and the new accumulator value. This step is repeated until the beginning of the list is reached; then the accumulator value is returned. `Proc` is always called in the same dynamic environment as `fold-right` itself.

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

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

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

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

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

**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. An implementation may check whether `combine` is an appropriate argument before applying it.

\[
\text{(remv 1 '(3 1 4 1 5 9 2 6 5))} \\
\quad \Rightarrow \text{(3 4 5 9 2 6 5)}
\]

\[
\text{(remv 'foo '(bar foo baz))} \\
\quad \Rightarrow \text{(bar baz)}
\]

**Implementation responsibilities:** The implementation must check the restrictions on `proc` to the extent performed by applying it as described. An implementation may check whether `proc` is an appropriate argument before applying it.

\[
\text{(memv proc list)} \\
\quad \text{procedure}
\]

\[
\text{(member obj list)} \\
\quad \text{procedure}
\]

\[
\text{(memq obj list)} \\
\quad \text{procedure}
\]

Proc should accept one argument and return a single value. Proc should not mutate *list*.

Each of these procedures returns a list of the elements of *list* that do not satisfy a given condition. The `remv` procedure applies `proc` to each element of *list* and returns a list of the elements of *list* for which `proc` returned #f. `Proc` is always called in the same dynamic environment as `remv` itself. The `remove`, `remv`, and `remq` procedures return a list of the elements that are not *obj*. The `remq` procedure uses `eq?` to compare *obj* with the elements of *list*, while `remv` uses `eqv?` and `remove` uses `equal?`. The elements of the result list are in the same order as they appear in the input list. If multiple returns occur from `remv`, the return values returned by earlier returns are not mutated.

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

\[
\text{(remove 1 '(3 1 4 1 5 9 2 6 5))} \\
\quad \Rightarrow \text{(3 1 4 5 9 2 6 5)}
\]

\[
\text{(memq 'a '(a b c))} \\
\quad \Rightarrow \text{(a b c)}
\]

\[
\text{(memq 'b '(a b c))} \\
\quad \Rightarrow \text{(b c)}
\]

\[
\text{(memq 'c '(a b c))} \\
\quad \Rightarrow \text{#f}
\]

\[
\text{(memq (list 'a) '(b (a) c))} \\
\quad \Rightarrow \text{#f}
\]

\[
\text{(member (list 'a) 
        '(b (a) c))} \\
\quad \Rightarrow \text{((a) c)}
\]

\[
\text{(memq 101 '(100 101 102))} \\
\quad \Rightarrow \text{unspecified}
\]

\[
\text{(memv 101 '(100 101 102))} \\
\quad \Rightarrow \text{(101 102)}
\]

**Implementation responsibilities:** The implementation must check that *list* is a chain of pairs up to the found element, or that in 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. An implementation may check whether `proc` is an appropriate argument before applying it.

\[
\text{(assp proc alist)} \\
\quad \text{procedure}
\]

\[
\text{(assoc obj alist)} \\
\quad \text{procedure}
\]
(assv obj alist) procedure
(assq obj alist) procedure

A list (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 condition, then #f is returned. The assp procedure successively applies proc to the car fields of alist and looks for a pair for which it returns a true value. Proc is always called in the same dynamic environment as assp itself. The assoc, assv, and assq procedures look for a pair that has obj as its car. The assoc procedure uses equal? to compare obj with the car fields of the pairs in alist, while assv uses eqv? and assq uses eq?.

Implementation responsibilities: The implementation must check that alist is a chain of pairs containing pairs up to n. If called with only one argument, Proc returns that argument.

(cons* obj1 ... objn obj) procedure
(cons* obj) procedure

If called with at least two arguments, cons* returns a freshly allocated chain of pairs whose cars are obj1, ..., objn, and whose last cdr is obj. If called with only one argument, cons* returns that argument.

4. Sorting

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 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. If multiple returns occur from list-sort or vector-sort, the return values returned by earlier returns are not mutated.

(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.

Implementation responsibilities: The implementation must check the restrictions on proc to the extent performed by applying it as described. An implementation may check whether proc is an appropriate argument before applying it.

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

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

(define e '((a 1) (b 2) (c 3)))

(assq 'a e) ⇒ (a 1)
(assq 'b e) ⇒ (b 2)
(assq 'd e) ⇒ #f
(assq (list 'a) '((a)) ((b)) ((c))) ⇒ #f
(assoc (list 'a) '((a)) ((b)) ((c))) ⇒ ((a))
(assoc (list 'a) '((a)) ((b)) ((c))) ⇒ #f
(assoc (list 'a) '((a)) ((b)) ((c))) ⇒ ((a))
(assq 5 '((2 3) (5 7) (11 13))) ⇒ unspecified
(assv 5 '((2 3) (5 7) (11 13))) ⇒ (5 7)

(define v (vector 3 5 2 1))
(vector-sort! < v) ⇒ unspecified
v ⇒ #(1 2 3 5)
whether proc is an appropriate argument before applying it.

5. Control structures

This chapter describes the (rnrs control (6)) library, which provides useful control structures.

\[
\text{when} \ (\text{test}) \ (\text{expression}_1) \ (\text{expression}_2) \ \ldots \ \text{syntax}
\]

\[
\text{unless} \ (\text{test}) \ (\text{expression}_1) \ (\text{expression}_2) \ \ldots \ \text{syntax}
\]

**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.

The final (expression) is in tail context if the when or unless form is itself in tail context.

\[
\begin{align*}
\text{when} \ (> \ 3 \ 2) \ &\text{ 'greater} \quad \Rightarrow \text{greater} \\
\text{when} \ (< \ 3 \ 2) \ &\text{ 'greater} \quad \Rightarrow \text{unspecified} \\
\text{unless} \ (> \ 3 \ 2) \ &\text{ 'less} \quad \Rightarrow \text{unspecified} \\
\text{unless} \ (< \ 3 \ 2) \ &\text{ 'less} \quad \Rightarrow \text{less}
\end{align*}
\]

The when and unless expressions are derived forms. They could be defined by the following macros:

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

```
(define-syntax unless
  (syntax-rules ()
    ((unless test result1 result2 ...) (if (not test) (begin result1 result2 ...))))))
```

\[
\text{do} \ ((\text{variable}_1) \ (\text{init}_1) \ (\text{step}_1) \ \ldots)
\]

```
(do ((\text{variable}_1) \ (\text{init}_1) \ (\text{step}_1) \ \ldots)
    ((\text{test}) \ (\text{expression}) \ \ldots)
    \ (\text{command}) \ \ldots)
```

**Syntax:** The (variable)/s, (init)/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 do expression returns unspecified values.

The region of the binding of a (variable) consists of the entire do expression except for the (init)/s.

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 [11.20] i.e., the last (expression) is also in a tail context.

\[
\text{do} \ ((\text{vec} \ (\text{make-vector} \ 5))
\]

\[
\begin{align*}
(i \ 0 \ (+ \ i \ 1)) \\
(= \ i \ 5 \ \text{vec})
\end{align*}
\]

\[
\begin{align*}
(\text{vector-set!} \ \text{vec} \ i \ i) & \quad \Rightarrow \ #(0 \ 1 \ 2 \ 3 \ 4) \\
\text{let} \ ((x '\(1 \ 3 \ 5 \ 7 \ 9))) & \quad \Rightarrow \ #(0 \ 1 \ 2 \ 3 \ 4) \\
\begin{align*}
(\text{do} \ ((\text{x} \ \text{x} \ \text{(cdr} \ \text{x}))) \\
(\text{sum} \ 0 \ (+ \ \text{sum} \ (\text{car} \ \text{x}))) \\
(\text{null?} \ \text{x} \ \text{sum})) & \quad \Rightarrow \ 25
\end{align*}
\]

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

```
(define-syntax do
  (syntax-rules ()
    ((do ((\text{var} \ \text{init} \ \text{step} \ \ldots) \ \ldots)
        \ (\text{test} \ \text{expr} \ \ldots)
        \ (\text{command} \ \ldots)
        \ (\text{letrec}
          ((\text{loop}
            \ (\text{lambda} \ (\text{var} \ \ldots)
              \ (\text{if} \ \text{test}
                \ (\text{begin}
                  \ #f \ ; \ \text{avoid empty begin}
                  \ \text{expr} \ \ldots)
                \ (\begin{align*}
                \ (\text{begin} \\
                \ \text{command} \\
                \ \ldots
                \end{align*})
                \ (\text{loop} \ (\text{do} \ \text{"step"} \ \text{var} \ \text{step} \ \ldots)
                \ldots))))))))
```

6. Records

This section describes abstractions for creating new data types representing records.

A record is a compound data structure with a fixed number of components, called fields. Each record has an associated type specified by a record-type descriptor, which is an object that specifies the fields of the record and various other properties that all records of that type share. Record objects are created by a record constructor, a procedure that creates a fresh record object and initializes its fields to values. Records of different types can be distinguished from each other and from other types of objects by record predicates. A record predicate returns #t when passed a record of the type specified by the record-type descriptor and #f otherwise. An accessor extracts from a record the component associated with a field, and a mutator changes the component to a different value.

Record types can 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. If a record type \( t \) extends another record type \( p \), each record of type \( t \) is also a record of type \( p \), and the predicate, accessors, and mutators applicable to a record of type \( p \) are also applicable to a record of type \( t \).

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.
A record type can be sealed to prevent it from being extended. Moreover, a record type can be nongenerative, i.e., it is globally identified by a “uid”, and new, compatible definitions of a nongenerative record type with the same uid as a previous always yield the same record type.

The record mechanism spans three libraries:

- the (rnrs records syntactic (6)) library, a syntactic layer for defining a record type and associated constructor, predicate, accessor, and mutators,
- the (rnrs records procedural (6)) library, a procedural layer for creating and manipulating record types and creating constructors, predicates, accessors, and mutators;
- the (rnrs records inspection (6)) library, a set of inspection procedures.

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.

Any of the standard types mentioned in this report may or may not be implemented as an opaque record type. Thus, it may be possible to use inspection on objects of the standard types.

The procedural layer 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 record operations provided through the procedural layer may, however, be less efficient than the operations provided through the syntactic layer, which is designed to allow expand-time determination of record-instance sizes and field offsets. Therefore, alternative implementations of syntactic record-type definition should, when possible, expand into the syntactic layer rather than the procedural layer.

The syntactic layer is used more commonly and therefore described first. This chapter uses the rtd and constructor-descriptor parameter names for arguments that must be record-type descriptors and constructor descriptors, respectively (see section [6.3]).

6.1. Mutability and equivalence of records

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 of a mutable record type returns a new record with a fresh location (see report section 5.10). Consequently, for two records obj1 and obj2, the return value of (eqv? obj1 obj2), as well as the return value of (eqv? obj1 obj2), adheres to the following criteria (see report section 11.5):

- If obj1 and obj2 have different record types (i.e., their record-type descriptors are not eqv?), eqv? returns #f.
- If obj1 and obj2 are both records of the same mutable record type, and are the results of two separate calls to record constructors, then eqv? returns #f.
- If obj1 and obj2 are both the result of a single call to a record constructor, then eqv? returns #t.
- If obj1 and obj2 are both records of the same record type, where applying an accessor to both yields results for which eqv? returns #f, then eqv? returns #f.

6.2. Syntactic layer

The syntactic layer is provided by the (rnrs records syntactic (6)) library. Some details of the specification are explained in terms of the specification of the procedural layer below.

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

(define-record-type (name spec) (record clause)*)

A define-record-type form defines a record type along with associated constructor descriptor and constructor, predicate, field accessors, and field mutators. The define-record-type form expands into a set of definitions in the environment where define-record-type appears; hence, it is possible to refer to the bindings (except for that of the record type itself) recursively.
The (name spec) specifies the names of the record type, constructor, and predicate. It must take one of the following forms:

((record name) (constructor name) (predicate name))
(record name)

(Record name), (constructor name), and (predicate name) must all be identifiers.

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

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

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

The second form of (name spec) is an abbreviation for the first form, where the name of the constructor is generated by prefixing the record name with make-, and the predicate name is generated by adding a question mark (?) to the end of the record name. For example, if the record name is frob, the name of the constructor is make-frob, and the predicate name is frob?.

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

(fields (field spec)*)

Each (field spec) has one of the following forms

(immutable (field name) (accessor name))
(mutable (field name))
(immutable (field name))
(mutable (field name))
(field name)

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

If (field spec) takes the third or fourth 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 -set! suffix to the accessor name. For example, if the record name is frob and the field name is widget, the accessor name is frob-widget and the mutator name is frob-widget-set!.

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

The (field name)s become, as symbols, the names of the fields in the record-type descriptor being created, in the same order.

The fields clause may be absent; this is equivalent to an empty fields clause.

(parent (parent name))

Specifies that the record type is to have parent type (parent name), where (parent name) is the (record name) of a record type previously defined using define-record-type. The record-type definition associated with (parent name) must not be sealed.

(protocol (expression))

(Expression) is evaluated in the same environment as the define-record-type form. It must evaluate to a procedure, and this procedure should be a protocol appropriate for the record type being defined.

The protocol is used to create a record-constructor descriptor as described below. If no protocol clause is specified, a constructor descriptor is still created using a default protocol. The clause can be absent only if the record type being 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, i.e., no extensions of the record type can be created. 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. See the specification of record-rtd below for details.

(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 record-type definitions should be equivalent, i.e., the implied arguments to
make-record-type-descriptor must be equivalent as described under make-record-type-descriptor. See section 6.3. If this condition is not met, it is either considered a syntax violation or an exception with condition type kassertion 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:

\[
\begin{align*}
\text{(let ((f (lambda (x) (define-record-type r ...))})
}\end{align*}
\]

\[
\begin{align*}
\text{(if x r? (make-r ...))))}
\end{align*}
\]

\[
\begin{align*}
\text{((f #t) (f #f))} \Rightarrow \#f
\end{align*}
\]

\[
\begin{align*}
\text{(parent-rtd (parent rtd)} \langle \text{parent cd} \rangle)
\end{align*}
\]

Specifies that the record type is to have its parent type specified by (parent rtd), which should be an expression evaluating to a record-type descriptor or #f, and (parent cd), which should be an expression evaluating to a constructor descriptor (see below) or #f.

If (parent rtd) evaluates to #f, then if (parent cd) evaluates to a value, that value must be #f.

If (parent rtd) evaluates to a record-type descriptor, the record type must not be sealed. Moreover, a record-type definition must not have both a parent and a parent-rtd clause.

Note: The syntactic layer is designed to allow record-instance sizes and field offsets to be determined at expand time, i.e., by a macro definition of define-record-type, as long as the parent (if any) is known. Implementations that take advantage of this may generate less efficient constructor, accessor, and mutator code when the parent-rtd clause is used, since the type of the parent is generally not known until run time. The parent clause should therefore be used instead when possible.

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.

If no parent clause is present, no parent-rtd clause is present, or a parent-rtd clause is present but (parent rtd) evaluates to #f, the record type is a base type.

The constructor created by a define-record-type form is a procedure as follows:

- If the record type is a base type and no protocol clause is present, the constructor accepts as many arguments as there are fields, in the same order as they appear in the fields clause, and returns a record object with the fields initialized to the corresponding arguments.

- If the record type is a base type and a protocol clause is present, the protocol expression, if it evaluates to a value, must evaluate to a procedure, and this procedure should accept a single argument. The protocol procedure is called once during the evaluation of the define-record-type form with a procedure p as its argument. It should return a procedure, which will become the constructor bound to (constructor name). The procedure p accepts as many arguments as there are fields, in the same order as they appear in the fields clause, and returns a record object with the fields initialized to the corresponding arguments.

The constructor returned by the protocol procedure can accept an arbitrary number of arguments, and should call p once to construct a record object, and return that record object.

For example, the following protocol expression for a record-type definition with three fields creates a constructor that accepts values for all fields, and initialized them in the reverse order of the arguments:

\[
\begin{align*}
\text{(lambda (p)}
\end{align*}
\]

\[
\begin{align*}
\text{(lambda (v1 v2 v3)}
\end{align*}
\]

\[
\begin{align*}
\text{(p v3 v2 v1))}
\end{align*}
\]

- If the record type is not a base type and a protocol clause is present, then the protocol procedure is called once with a procedure n as its argument. As in the previous case, the protocol procedure should return a procedure, which will become the constructor bound to (constructor name). However, n is different from p in the previous case: It accepts arguments corresponding to the arguments of the constructor of the parent type. It then returns a procedure p that accepts as many arguments as there are (additional) fields in this type, in the same order as in the fields clause, and returns a record object with the fields of the parent record types initialized according to their constructors and the arguments to n, and the fields of this record type initialized to its arguments of p.

The constructor returned by the protocol procedure can accept an arbitrary number of arguments, and should call n once to construct the procedure p, and call p once to create the record object, and finally return that record object.

For example, the following protocol expression assumes that the constructor of the parent type takes three arguments:

\[
\begin{align*}
\text{(lambda (n)}
\end{align*}
\]

\[
\begin{align*}
\text{(lambda (v1 v2 v3 x1 x2 x3 x4))}
\end{align*}
\]

\[
\begin{align*}
\text{(let ((p (n v1 v2 v3)))}
\end{align*}
\]

\[
\begin{align*}
\text{(p x1 x2 x3 x4))})}
\end{align*}
\]

The resulting constructor accepts seven arguments, and initializes the fields of the parent types according to the constructor of the parent type, with v1, v2,
and v3 as arguments. It also initializes the fields of this record type to the values of x1, ..., x4.

• If there is a parent clause, but no protocol clause, then the parent type must not have a protocol clause itself. Similarly, if there is a parent-rtd clause whose (parent rtd) evaluates to a record-type descriptor, but no protocol clause, then the (parent cd) expression, if it evaluates to a value, must evaluate to #f. The constructor bound to (constructor name) is a procedure that accepts arguments corresponding to the parent types’ constructor first, and then one argument for each field in the same order as in the fields clause. The constructor returns a record object with the fields initialized to the corresponding arguments.

A protocol may perform other actions consistent with the requirements described above, including mutation of the new record or other side effects, before returning the record.

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 (p)
      (lambda (n) (p (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 (p)
      (lambda (n) (p (make-widget n))))))
```

In the following example uses the record? procedure from the (rnrs records inspection (6)) library (section 6.4):

```
(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 (n)
      ((n x y c)
       (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) ⇒ unspecified
(point-y p1) ⇒ 17)
```
6.3. Procedural layer

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

\begin{verbatim}
(make-record-type-descriptor name procedure parent uid sealed? opaque? fields)
\end{verbatim}

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

The \texttt{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 \texttt{parent} argument must be either \texttt{#f} or an \texttt{rtd}. If it is an \texttt{rtd}, the returned record type, \texttt{t}, extends the record type \texttt{p} represented by \texttt{parent}. An exception with condition type \texttt{&assertion} is raised if \texttt{parent} is sealed (see below).

The \texttt{uid} argument must be either \texttt{#f} or a symbol. If \texttt{uid} is a symbol, the record-creation operation is non-generative 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} argument.

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 the parents are not opaque. 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.

\textit{Note:} Users are encouraged to use symbol names constructed using the UUID namespace \cite{uuid} (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 with 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 fields argument must be a vector of field specifiers. Each field specifier must be a list of the form (mutable name) or a list of the form (immutable name). Each name must be a symbol and names the corresponding field of the record type; the names need not be distinct. A field identified as mutable may be modified, whereas, 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 fields is modified after \texttt{make-record-type-descriptor} has been called, the effect on the returned rtd is unspecified.

A generative record-type descriptor created by a call to \texttt{make-record-type-descriptor} is not 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 eqv? only to itself, i.e., (eqv? rtd1 rtd2) iff (eqv? rtd1 rtd2). Also, two nongenerative record-type descriptors are 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

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

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

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 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 #f. If it is #f, a default protocol procedure is supplied.

If protocol is a procedure, it is handled analogously to the protocol expression in a \texttt{define-record-type} form.

If rtd is a base record type parent-constructor-descriptor must be #f. In this case, protocol is called by \texttt{record-constructor} with a single argument \texttt{p}. \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 protocol should call \texttt{p} once with the number of arguments \texttt{p} 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 (\texttt{lambda (p) p}). Note that the procedure returned by protocol is otherwise unconstrained; specifically, it can take any number of arguments.

If rtd is an extension of another record type parent-rtd and protocol is a procedure, parent-constructor-descriptor must be a constructor descriptor of parent-rtd or #f. If parent-constructor-descriptor is a constructor descriptor, protocol is called by \texttt{record-constructor} with a single argument \texttt{n}, which is a procedure that accepts the same number of arguments as the constructor of parent-constructor-descriptor and returns a procedure \texttt{p} that, when called, constructs the record itself. The \texttt{p} 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 parent-rtd and its parents initialized as specified by parent-constructor-descriptor.

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

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

This passes arguments \texttt{v1}, \texttt{v2}, \texttt{v3} to \texttt{n} for parent-constructor-descriptor and calls \texttt{p} 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 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 accepted by the parent constructor.

\texttt{Protocol} may be #f, specifying a default constructor that accepts one argument for each field of \texttt{rtd} (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 (\texttt{lambda (p) p}). If \texttt{rtd} is an extension of another type, then parent-constructor-descriptor must be either #f or itself specify a default constructor, and the default protocol procedure behaves as if it were:

\begin{verbatim}
(lambda (n) (lambda (v1 ... vj x1 ... xk) (let ((p (n v1 ... vj))) (p 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, \( j \) is the complete number of fields of the parent type, and \( k \) is the number of fields of \( rtd \) itself.)

If \( rtd \) is an extension of another record type and \( \text{parent-constructor-descriptor} \) is \( \#f \), \( \text{parent-constructor-descriptor} \) is treated as if it were a constructor descriptor for the parent \( rtd \) of \( rtd \) with a default protocol.

**Implementation responsibilities:** If \( \text{protocol} \) 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. An implementation may check whether \( \text{protocol} \) is an appropriate argument before applying it.

```scheme
(define rtd1
  (make-record-type-descriptor
   'rtd1 #f #f #f #f
   '((immutable x1) (immutable x2))))

(define rtd2
  (make-record-type-descriptor
   'rtd2 rtd1 #f #f #f
   '((immutable x3) (immutable x4))))

(define rtd3
  (make-record-type-descriptor
   'rtd3 rtd2 #f #f #f
   '((immutable x5) (immutable x6))))

(define protocol1
  (lambda (p)
    (lambda (a b c)
      (p (+ a b) (+ b c))))))

(define protocol2
  (lambda (n)
    (lambda (a b c  d e f)
      (let ((p (n a b c)))
        (p (+ d e) (+ e f))))))

(define protocol3
  (lambda (n)
    (lambda (a b c  d e f g h i)
      (let ((p (n a b c  d e f g h i)))
        (p (+ g h) (+ h i)))))))

(define cd1
  (make-record-constructor-descriptor
   rtd1 #f protocol1))

(define cd2
  (make-record-constructor-descriptor
   rtd2 cd1 protocol2))

(define cd3
  (make-record-constructor-descriptor
   rtd3 cd2 protocol3))

(define make-rtd1 (record-constructor cd1))
(define make-rtd2 (record-constructor cd2))
(define make-rtd3 (record-constructor cd3))

(make-rtd3 1 2 3 4 5 6 7 8 9)
⇒
(record with fields initialized to 3, 5, 9, 11, 15, 17)
```

**record-predicate** \( rtd \)

Procedure

Returns a procedure that, given an object \( \text{obj} \), returns \#t if \( \text{obj} \) is a record of the type represented by \( rtd \), and \#f otherwise.

**record-accessor** \( rtd \) \( k \)

Procedure

\( K \) must be a valid field index of \( rtd \). The **record-accessor** procedure returns a one-argument procedure whose argument must be a record of the type represented by \( rtd \). This procedure returns the value of the selected field of that record.

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

**record-mutator** \( rtd \) \( k \)

Procedure

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

```scheme
(define :point
  (make-record-constructor-descriptor
   rtd3 cd2 protocol3))
```

```scheme
(define make-rtd1 (record-constructor cd1))
(define make-rtd2 (record-constructor cd2))
(define make-rtd3 (record-constructor cd3))

(make-rtd3 1 2 3 4 5 6 7 8 9)
⇒
(record with fields initialized to 3, 5, 9, 11, 15, 17)
```
6. Records

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

(define make-point (record-constructor :point-cd))

(define point? (record-predicate :point))

(define point-x (record-accessor :point 0))

(define point-y (record-accessor :point 1))

(define point-x-set! (record-mutator :point 0))

(define point-y-set! (record-mutator :point 1))

(define p1 (make-point 1 2))

(point? p1) => #t

(point-x p1) => 1

(point-y p1) => 2

(point-x-set! p1 5) => unspecified

(point-x p1) => 5

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

(define make-point2
  (record-constructor
   (make-record-constructor-descriptor :point2 #f #f)))

(define point2? (record-predicate :point2))

(define point2-xx (record-accessor :point2 0))

(define point2-yy (record-accessor :point2 1))

(define p2 (make-point2 1 2 3 4))

(point? p2) => #t

(point-x p2) => 1

(point-y p2) => 2

(point2-xx p2) => 3

(point2-yy p2) => 4

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

(define make-point-abs
  (record-constructor
   :point-cd/abs
   (lambda (p)
     (lambda (x y c)
       ((p x y) (color->rgb c))))))

(define cpoint-rgb
  (record-accessor :cpoint 0))

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

(cpoint-rgb (make-point/abs -1 -3 'red))

(point-x (make-point/abs -1 -3 'red)) => -1

(point-x (make-point/abs -1 -3 'red)) => 1

6.4. Inspection

The (rnrs records inspection (6)) library provides procedures 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.

(record? obj) procedure

Returns #t if obj is a record, and its record type is not opaque, and 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 `#t` if the record-type descriptor is sealed, and `#f` if not.

`record-type-opaque? rtd` procedure

Returns `#t` if the the record-type descriptor is opaque, and `#f` if not.

`record-type-field-names rtd` procedure

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

`record-field-mutable? rtd k` procedure

Returns `#t` if the field specified by `k` of the type represented by `rtd` is mutable, and `#f` if not. $K$ is as in `record-accessor`.

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.

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 5.12.

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 and should accept zero arguments. The `with-exception-handler` procedure returns the results of invoking `thunk` without arguments. `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 `thunk` to the extent performed by applying it as described above. The implementation must
check the restrictions on \texttt{handler} to the extent performed by applying it as described when it is called as a result of a call to \texttt{raise} or \texttt{raise-continuable}. An implementation may check whether \texttt{handler} is an appropriate argument before applying it.

\begin{verbatim}
(guard ((variable) syntax
  ⟨cond clause1⟩ ⟨cond clause2⟩ ...) (body)) => auxiliary syntax
else auxiliary syntax
\end{verbatim}

\textit{Syntax:} Each ⟨cond clause⟩ is as in the specification of \texttt{cond}. (See report section \[11.4.5\] \Rightarrow and \texttt{else} are the same as in the \texttt{(rnrs base (6))} library.

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

The final expression in a ⟨cond clause⟩ is in a tail context if the \texttt{guard} expression itself is.

\begin{verbatim}
(raise \textit{obj}) procedure
\end{verbatim}

Raises a non-continuable exception by invoking the current exception handler on \textit{obj}. The handler is called with a continuation whose dynamic environment is that of the call to \texttt{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 \texttt{&non-continuable} is raised in the same dynamic environment as the handler.

\begin{verbatim}
(raise-continuable \textit{obj}) procedure
\end{verbatim}

 Raises a \textit{continuable exception} by invoking the current exception handler on \textit{obj}. The handler is called with a continuation that is equivalent to the continuation of the call to \texttt{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 \texttt{raise-continuable}.

\begin{verbatim}
(raise (condition
  (make-error)
  (make-message-condition "I am an error")))
prints: I am an error \Rightarrow error
\end{verbatim}
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. Scheme also defines a number of basic condition types.

Scheme conditions provides two mechanisms to enable communication about an 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 11.1. 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 with itself as 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 has no fields and is neither sealed nor opaque.

(condition condition1 ...) procedure

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 condition2, 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) procedure

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.

Note: Because condition decomposes its arguments into simple conditions, simple-conditions always returns a “flattened” list of simple conditions.

(condition? obj) procedure

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

(condition-predicate rtd) procedure

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, and #f otherwise.

(condition-accessor rtd proc) procedure

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)))
7. Exceptions and conditions

(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 bar))
⇒ unspecified

(real-cond1? (condition foo))
⇒ unspecified

(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))
  (list foo))
⇒ #t

Syntax: (Condition-type), (supertype), (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 6.2). 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 6.3): 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) ⇒ #t
(c1? v4) ⇒ #t
(c2? v4) ⇒ #t
(c-x v4) ⇒ "V1"
(c1-a v4) ⇒ "a1"
(c2-b v4) ⇒ "b2"

(define v5 (condition v2 v3))

(c? v5) ⇒ #t
(c1? v5) ⇒ #t
(c2? v5) ⇒ #t
(c-x v5) ⇒ "V2"
(c1-a v5) ⇒ "a3"
(c2-b v5) ⇒ "b2"
### 7.3. Standard condition types

#### &message

- **(make-message-condition message)**
- **(message-condition? obj)**
- **(condition-message condition)**

This condition type could be defined by

```scheme
(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

- **(make-warning)**
- **(warning? obj)**

This condition type could be defined by

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

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

#### &serious

- **(make-serious-condition)**
- **(serious-condition? obj)**

This condition type could be defined by

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

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.

#### &error

- **(make-error)**
- **(error? obj)**

This condition type could be defined by

```scheme
(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

- **(make-violation)**
- **(violation? obj)**

This condition type could be defined by

```scheme
(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.

#### &assertion

- **(make-assertion-violation)**
- **(assertion-violation? obj)**

This condition type could be defined by

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

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

#### &irritants

- **(make-irritants-condition irritants)**
- **(irritants-condition? obj)**
- **(condition-irritants condition)**

This condition type could be defined by

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

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.
Irritants should be 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 11.14.

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

This condition type could be defined by

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

Who should be 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 11.14), and the syntax-violation procedure (section 12.9).

\[
\text{&non-continuable} \\
\text{(make-non-continuable-violation)} \\
\text{(non-continuable-violation? obj)}
\]

This condition type could be defined by

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

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

\[
\text{&implementation-restriction} \\
\text{(make-implementation-restriction-violation)} \\
\text{(implementation-restriction-violation? obj)}
\]

This condition type could be defined by

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

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 11.3)

\[
\text{&lexical} \\
\text{(make-lexical-violation)} \\
\text{(lexical-violation? obj)}
\]

This condition type could be defined by

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

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

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

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. Form should be the erroneous syntax object or a datum representing the code of the erroneous form. Subform should be 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.

\[
\text{&undefined} \\
\text{(make-undefined-violation)} \\
\text{(undefined-violation? obj)}
\]

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.

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 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.

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.

\( \&i/o \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-error</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-error? obj</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

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

\( \&i/o-read \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-read-error</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-read-error? obj</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

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

\( \&i/o-write \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-write-error</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-write-error? obj</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

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

\( \&i/o-invalid-position \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-invalid-position-error position</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-invalid-position-error? obj</td>
<td>procedure</td>
</tr>
<tr>
<td>(i/o-error-position condition)</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

This condition type describes attempts to set the file position to an invalid position. Position should be the file position that the program intended to set. This condition describes a range error, but not an assertion violation.

\( \&i/o-filename \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-filename-error filename</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-filename-error? obj</td>
<td>procedure</td>
</tr>
<tr>
<td>(i/o-error-filename condition)</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

This condition type describes an I/O error that occurred during an operation on a named file. Filename should be the name of the file.

\( \&i/o-file-protection \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-file-protection-error filename</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-file-protection-error? obj</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

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

\( \&i/o-file-is-read-only \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-file-is-read-only-error filename</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-file-is-read-only-error? obj</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

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

\( \&i/o-file-already-exists \)

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-i/o-file-already-exists-error filename</td>
<td>procedure</td>
</tr>
<tr>
<td>i/o-file-already-exists-error? obj</td>
<td>procedure</td>
</tr>
</tbody>
</table>

This condition type could be defined by

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

A condition of this type specifies that an operation tried to operate on a named file that already exists. Filename should be the name of the file.
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.

\[
\text{&i/o-file-does-not-exist condition type}
\]

\[
\text{(make-&i/o-file-does-not-exist-error filename)}
\]

procedure

\[
(i/o-file-does-not-exist-error? obj)
\]

procedure

This condition type could be defined by

\[
 \text{(define-condition-type &i/o-file-does-not-exist &i/o-filename make-i/o-file-does-not-exist-error i/o-file-does-not-exist-error?)}
\]

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

\[
\text{&i/o-port condition type}
\]

\[
\text{(make-&i/o-port-error pobj)}
\]

procedure

\[
(i/o-port-error? obj)
\]

procedure

\[
(i/o-error-port condition)
\]

procedure

This condition type could be defined by

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

This condition type specifies the port with which an I/O error is associated. \(Pobj\) should be the port. Conditions raised by procedures accepting a port as an argument should include an \&i/o-port-error condition.

8.2. Port I/O

The \(\text{(rnrs io ports (6))}\) 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 \(\text{(rnrs io ports (6))}\) 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 \(\text{(rnrs io ports (6))}\) library also distinguishes between binary ports, which are sources or sinks for uninterpreted bytes, and textual ports, which are sources or sinks for characters and strings.

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

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 operating 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 [4]) over the symbols constituting valid file options. A file-options parameter name means that the corresponding argument must be a file-options object.

\[
\text{(file-options (file-options symbol) \ldots)}
\]

syntax

Each (file-options symbol) must be a symbol. The file-options syntax returns a file-options object that encapsulates the specified options.

When supplied to an operation that opens a file for output, the file-options object returned by (file-options) specifies that the file is created if it does not exist and an exception 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.

- \text{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.

- \text{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.

- \text{no-truncate} If the file already exists and the exception 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 current position is still set to the beginning of the file.

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

Note: Only the name of `(file-options symbol)` is significant.

### 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 and reading up to line endings, or other implementation-dependent behavior, 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 as a read operation.

```scheme
(buffer-mode (buffer-mode symbol))
```

**(buffer-mode symbol)** must be a symbol whose name is one of `none`, `line`, and `block`. The result is the corresponding symbol, and specifies the associated buffer mode.

Note: Only the name of (buffer-mode symbol) is significant.

```scheme
(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 [12]. 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). 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.

```scheme
(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 [12].

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.

```scheme
(eol-style (eol-style symbol))
```

(Eol-style symbol) should be a symbol whose name is one of `lf`, `cr`, `crlf`, `nel`, `crnel`, `ls`, and `none`. The form evaluates to the corresponding symbol. If the name of eol-style symbol is not one of these symbols, the effect and result are implementation-dependent; in particular, the result may be an eol-style symbol acceptable as an eol-style 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.

*Note:* Only the name of `(error-handling-style symbol)` is significant.

```lisp
(native-eol-style) procedure
```

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

```lisp
(k/i/o-decoding) condition type
(make-i/o-decoding-error pobj) procedure
(i/o-decoding-error? obj) procedure
```

This condition type could be defined by

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

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.

When such an exception is raised, the port’s position is past the invalid encoding.

```lisp
(k/i/o-encoding) condition type
(make-i/o-encoding-error pobj objc) procedure
(i/o-encoding-error? obj) procedure
(i/o-encoding-error-char condition) procedure
```

This condition type could be defined by

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

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. `objc` should be the character that could not be encoded.

```lisp
(error-handling-mode (error-handling-mode symbol)) syntax
```

`(error-handling-mode symbol)` should be a symbol whose name is one of `ignore`, `raise`, and `replace`. The form evaluates to the corresponding symbol. If `(error-handling-mode symbol)` is not one of these identifiers, effect and result are implementation-dependent:

The result may be an error-handling-mode symbol acceptable as a `handling-mode` argument to `make-transcoder`. If it is not acceptable as a `handling-mode` argument to `make-transcoder`, an exception is raised.

*Note:* Only the name of `(error-handling-style symbol)` is significant.

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

If a textual input operation encounters an invalid or incomplete character encoding, and the error-handling mode is `ignore`, an appropriate number of bytes of the invalid encoding are ignored and decoding continues with the following bytes. If the error-handling mode is `replace`, the replacement character `U+FFFD` is injected into the data stream, an appropriate number of bytes are ignored, and decoding continues with the following bytes. If the error-handling mode is `raise`, an exception with condition type `k/i/o-decoding` is raised.

If a textual output operation encounters a character it cannot encode, and the error-handling mode is `ignore`, the character is ignored and encoding continues with the next character. If the error-handling mode is `replace`, a codec-specific replacement character is emitted by the transcoder, and encoding continues with the next character. The replacement character is `U+FFFD` for transcoders whose codec is one of the Unicode encodings, but is the `?` character for the Latin-1 encoding. If the error-handling mode is `raise`, an exception with condition type `k/i/o-encoding` is raised.

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

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

```lisp
(native-transcoder) procedure
```

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

(eqv? (eof-object) (eof-object)) ⇒ #t
(eq? (eof-object) (eof-object)) ⇒ #t

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

(eof-object? obj)  procedure
Returns #t if obj is the end-of-file object, #f otherwise.

8.2.6. Input and output ports

The operations described in this section are common to input and output ports, both binary and textual. A port may also have an associated position that specifies a particular place within its data sink or source, and may also provide operations for inspecting and setting that place.

(port? obj)  procedure
Returns #t if the argument is a port, and returns #f otherwise.

(port-transcoder port)  procedure
Returns the transcoder associated with port 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 binary-port. If 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 binary-port. If 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 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 binary-port, even though 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
(post-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 a value of some implementation-dependent type representing the port’s position; this value may be useful only as the pos argument to set-port-position!, if the latter is supported on the port (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.

(set-port-position! port)  procedure
The 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, and an exception with condition type &i/o-invalid-position if pos is not in the range of valid positions of port. 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-file-input/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 &i/o-invalid-position.

(close-port port) procedure

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

(call-with-port port proc) procedure

Proc must accept one argument. The call-with-port procedure calls proc with port as an argument. If proc returns, port is closed automatically and the values returned by proc are returned. If proc does not return, port is not closed automatically, except perhaps when it is possible to prove that 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 &i/o-decoding exception because the character encoding at the port’s position is incomplete up to the next end of file, a subsequent call to get-char may successfully decode a character if bytes completing the encoding are available after the end of file.

(input-port? obj) procedure

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

(port_eof? input-port) procedure

Returns #t if the lookahead-u8 procedure (if input-port is a binary port) or the lookahead-char procedure (if input-port is a textual port) would return the end-of-file object, and #f otherwise. The operation may block indefinitely if no data is available but the port cannot be determined to be at end of file.

(open-file-input-port filename) procedure

(open-file-input-port filename file-options) procedure

(open-file-input-port filename file-options buffer-mode) procedure

(open-file-input-port filename file-options maybe-transcoder) procedure

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

The open-file-input-port procedure returns an input port for the named file. The file-options and maybe-transcoder arguments are optional.

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 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 is implementation-dependent (and possibly transcoder-dependent).

(open-bytevector-input-port bytevector) procedure

(open-bytevector-input-port bytevector maybe-transcoder) procedure

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

The open-bytevector-input-port procedure returns an input port whose bytes are drawn from 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. The port should support the port-position and set-port-position! operations.

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

(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, the read! procedure should write no bytes and return 0.

- (get-position)
  The get-position procedure (if supplied) should return an exact integer object representing 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.

Implementation responsibilities: The implementation must 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-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.

- (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 should return an exact integer object
representing the number of characters that it has written. To indicate an end of file, 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 must 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.

Note: Even when the get-position procedure is supplied, the port-position procedure cannot generally return a precise value for a custom textual input port if data has been read from the port. Therefore, it is likely that this entry will change in a future version of the report.

8.2.8. Binary input

(get-u8 binary-input-port) procedure

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

(lookahead-u8 binary-input-port) procedure

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

(get-bytevector-n binary-input-port count) procedure

Count must be an exact, non-negative integer object representing the number of bytes to be read. 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, 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) procedure

Start and count must be exact, non-negative integer objects, with count 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 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-some binary-input-port) procedure

Reads from binary-input-port, blocking as necessary, until bytes are available from binary-input-port or until an end of file is reached. If bytes become available, get-bytevector-some returns a freshly allocated bytevector containing the initial available bytes (at least one), and it updates binary-input-port to point just past these bytes. If no input bytes are seen before an end of file is reached, the end-of-file object is returned.

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

Attempts to read all bytes 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 and updates binary-input-port to point just past these bytes. Otherwise, get-bytevector-all returns the end-of-file object.
The operation may block indefinitely waiting to see if more bytes will become available, even if some bytes are already available.

8.2.9. Textual input

(get-char textual-input-port) procedure

Reads from textual-input-port, blocking as necessary, until a complete character is available from textual-input-port, 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 character. If an end of file is reached before any character is read, get-char returns the end-of-file object.

(lookahead-char textual-input-port) procedure

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

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

(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.

The get-string-n procedure reads from textual-input-port, blocking as necessary, until count characters 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 characters read. If no characters can be read before an end of file, the end-of-file object is returned.

(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.

The get-string-n! procedure 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 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.

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

(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 characters have been read and decoded as characters, a string containing those characters is returned. If an end of file is encountered before any characters 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 8.2.3.

(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 4.2) 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 buffered 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 is 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 bytes (regardless of the port’s current position), removes the accumulated bytes from the port, and resets the port’s position.

(call-with-bytevector-output-port proc) procedure
(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 bytes written to it and calls proc with that output port as an argument. Whenever proc returns, a bytevector consisting of all the port’s accumulated bytes (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 open-bytevector-output-port procedure returns two values: an output port and an extraction procedure. The output port accumulates the bytes 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 is implementation-dependent (and possibly transcoder-dependent).

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.

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

Proc must accept one argument. The call-with-string-output-port procedure creates a textual output port that accumulates the characters written to it and calls proc with
that output port as an argument. Whenever proc returns, a string consisting of 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 port-position and set-port-position! operations.

(make-custom-binary-input-port id procedure write! get-position set-position! close)

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

Implementation responsibilities: The implementation must 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

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

Writes octet to the output port and returns unspecified values.

(pput-bytevector binary-output-port bytevector) procedure

(pput-bytevector binary-output-port bytevector start count) procedure

Start and count will be non-negative exact integer objects, and bytevector will be a bytevector whose length is at least start + count. The write! procedure should write up to count bytes from bytevector starting at index start to the byte sink. The write! procedure should return the number of bytes that it wrote, as an exact integer object.
Start and count must be non-negative exact integer objects that default to 0 and \( (\text{bytevector-length bytevector}) - \text{start} \), respectively. Bytevector must have a length of at least \( \text{start} + \text{count} \). 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

\begin{verbatim}
(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) 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 \( \text{start} + \text{count} \). Start defaults to 0. Count defaults to \( (\text{string-length string}) - \text{start} \). The put-string procedure writes the count characters of string starting at index start to the output port. The put-string procedure returns unspecified values.

(put-datum textual-output-port datum) procedure
Datum should be a datum value. The put-datum procedure writes an external representation of datum to textual-output-port. The specific external representation is implementation-dependent. However, whenever possible, an implementation should produce a representation for which get-datum, when reading the representation, will return an object equal (in the sense of equal?) to datum. Note: Not all datums may allow producing an external representation for which get-datum will produce an object that is equal to the original. Specifically, NaNs contained in datum may make this impossible.

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

### 8.2.13. Input/output ports

\begin{verbatim}
(open-file-input/output-port filename) procedure
(open-file-input/output-port filename file-options) procedure
(open-file-input/output-port filename file-options buffer-mode) procedure
\end{verbatim}

 República de Venezuela, Caracas, 2008.

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

\begin{verbatim}
(make-custom-binary-input/output-port procedure id read! write! get-position set-position! close)
\end{verbatim}

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

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

\begin{verbatim}
(make-custom-textual-input/output-port procedure id read! write! get-position set-position! close)
\end{verbatim}

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-textual-input-port and make-custom-textual-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-textual-input-port.

**Note:** Even when both get-position and set-position! procedures are supplied, the port-position procedure cannot generally return a precise value for a custom textual input/output port, and a put operation cannot precisely position the port for output, after data has been read from the port. Therefore, it is likely that this entry will change in a future version of the report.
8.3. Simple I/O

This section describes the \texttt{(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 revision of this report \cite{8}

The ports created by the procedures of this library are textual ports associated implementation-dependent transcoders.

\begin{verbatim}
(eof-object)                procedure
(eof-object? obj)          procedure
These are the same as \texttt{eof-object} and \texttt{eof-object?} from the \texttt{(rnrs io ports (6))} library.
\end{verbatim}

\begin{verbatim}
(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 \texttt{filename} for input or for output, with no specified file options, and call \texttt{proc} with the obtained port as an argument. If \texttt{proc} returns, the port is closed automatically and the values returned by \texttt{proc} are returned. If \texttt{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 \texttt{input-port?} and \texttt{output-port?} procedures in the \texttt{(rnrs io ports (6))} library.
\end{verbatim}

\begin{verbatim}
(current-input-port)       procedure
(current-output-port)      procedure
(current-error-port)       procedure
These are the same as the \texttt{current-input-port}, \texttt{current-output-port}, and \texttt{current-error-port} procedures from the \texttt{(rnrs io ports (6))} library.
\end{verbatim}

\begin{verbatim}
(with-input-from-file filename thunk)  procedure
(with-output-to-file filename thunk)   procedure
\texttt{Thunk} must be a procedure and must accept zero arguments. The file is opened for input or output using empty file options, and \texttt{thunk} is called with no arguments. During the dynamic extent of the call to \texttt{thunk}, the obtained port is made the value returned by \texttt{current-input-port} or \texttt{current-output-port} procedures; the previous default values are reinstated when the dynamic extent is exited. When \texttt{thunk} returns, the port is closed automatically. The values returned by \texttt{thunk} are returned. If an escape procedure is used to escape back into the call to \texttt{thunk} after \texttt{thunk} is returned, the behavior is unspecified.
\end{verbatim}

\begin{verbatim}
(open-input-file filename)      procedure
Opens \texttt{filename} for input, with empty file options, and returns the obtained port.
\end{verbatim}

\begin{verbatim}
(open-output-file filename)     procedure
Opens \texttt{filename} for output, with empty file options, and returns the obtained port.
\end{verbatim}

\begin{verbatim}
(close-input-port input-port)    procedure
(close-output-port output-port)  procedure
Closes \texttt{input-port} or \texttt{output-port}, respectively.
\end{verbatim}

\begin{verbatim}
(read-char)                  procedure
(read-char textual-input-port) procedure
Reads from \texttt{textual-input-port}, blocking as necessary until a character is available from \texttt{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, \texttt{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, \texttt{read-char} returns the end-of-file object.
If \texttt{textual-input-port} is omitted, it defaults to the value returned by \texttt{current-input-port}.
\end{verbatim}

\begin{verbatim}
(peek-char)                  procedure
(peek-char textual-input-port) procedure
This is the same as \texttt{read-char}, but does not consume any data from the port.
\end{verbatim}

\begin{verbatim}
(read)                      procedure
(read textual-input-port)   procedure
Reads an external representation from \texttt{textual-input-port} and returns the datum it represents. The \texttt{read} procedure operates in the same way as \texttt{get-datum}, see section \texttt{8.2.9}.
If \texttt{textual-input-port} is omitted, it defaults to the value returned by \texttt{current-input-port}.
\end{verbatim}

\begin{verbatim}
(write-char char)            procedure
(write-char char textual-output-port) procedure
Writes an encoding of the character \texttt{char} to the \texttt{textual-output-port}, and returns unspecified values.
\end{verbatim}
If `textual-output-port` is omitted, it defaults to the value returned by `current-output-port`.

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

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

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

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

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

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

Writes the external representation of `obj` to `textual-output-port`. The `write` procedure operates in the same way as `put-datum`; see section [8.2.12](#).

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

### 9. 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 [8.1](#).

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

`Filenname` must be a file name (see section [8.2.1](#)). The `file-exists?` procedure returns `#t` if the named file exists at the time the procedure is called, `#f` otherwise.

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

`Filenname` must be a file name (see section [8.2.1](#)). The `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 `&i/o-filename` is raised.

### 10. Command-line access and exit values

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

```
(command-line)  procedure
```

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.

```
(exit)  procedure
(exit obj)  procedure
```

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

### 11. 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.

#### 11.1. Bitwise operations

A number of procedures operate on the binary two’s-complement representations of exact integer objects: Bit positions within an exact integer object are counted from the right, i.e. bit 0 is the least significant bit. 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.

#### 11.2. Fixnums

Every implementation must define its fixnum range as a closed interval

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

such that \( w \) is a (mathematical) integer \( w \geq 24 \). Every mathematical integer within an implementation’s fixnum range must correspond to an exact integer 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 fixnums (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) procedure
Returns \#t if \texttt{obj} is an exact integer object within the fixnum range, \#f otherwise.

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

(fx=\? \texttt{fx1 \texttt{fx2 \texttt{fx3 ...}}}) procedure
(fx>\? \texttt{fx1 \texttt{fx2 \texttt{fx3 ...}}}) procedure
(fx<\? \texttt{fx1 \texttt{fx2 \texttt{fx3 ...}}}) procedure
(fx\>=\? \texttt{fx1 \texttt{fx2 \texttt{fx3 ...}}}) procedure
(fx\<=\? \texttt{fx1 \texttt{fx2 \texttt{fx3 ...}}}) procedure
These procedures return \#t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, \#f otherwise.

(fxzero? \texttt{fx}) procedure
(fpositive? \texttt{fx}) procedure
(fnegative? \texttt{fx}) procedure
(fodd? \texttt{fx}) procedure
(feven? \texttt{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.

(fxmax \texttt{fx1 \texttt{fx2 ...}}) procedure
(fxmin \texttt{fx1 \texttt{fx2 ...}}) procedure
These procedures return the maximum or minimum of their arguments.

(fx+ \texttt{fx1 \texttt{fx2}}) procedure
(fx* \texttt{fx1 \texttt{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 \texttt{&implementation-restriction} is raised if that sum or product is not a fixnum.

(fx- \texttt{fx1 \texttt{fx2}}) procedure
(fx- \texttt{fx}) procedure
With two arguments, this procedure returns the difference \(\texttt{fx1 - fx2}\), 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 \texttt{&implementation-restriction} is raised if the mathematically correct result of this procedure is not a fixnum.

\begin{verbatim}
(fx- (least-fixnum)) ⇒ \texttt{&implementation-restriction} exception
\end{verbatim}

(fxdiv-and-mod \texttt{fx1 \texttt{fx2}}) procedure
(fxdiv \texttt{fx1 \texttt{fx2}}) procedure
(fmodx \texttt{fx1 \texttt{fx2}}) procedure
(fxdiv0-and-mod0 \texttt{fx1 \texttt{fx2}}) procedure
(fxdiv0 \texttt{fx1 \texttt{fx2}}) procedure
(fxmod0 \texttt{fx1 \texttt{fx2}}) procedure
These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section \texttt{11.7.3}:

\begin{verbatim}
(fxdiv \texttt{fx1 \texttt{fx2}}) ⇒ \texttt{fx1 div fx2}
(fmodx \texttt{fx1 \texttt{fx2}}) ⇒ \texttt{fx1 mod fx2}
(fxdiv-and-mod \texttt{fx1 \texttt{fx2}}) ⇒ \texttt{fx1 div fx2, fx1 mod fx2}; two return values
(fxdiv0 \texttt{fx1 \texttt{fx2}}) ⇒ \texttt{fx1 div0 fx2}
(fmodx0 \texttt{fx1 \texttt{fx2}}) ⇒ \texttt{fx1 mod0 fx2}
(fxdiv0-and-mod0 \texttt{fx1 \texttt{fx2}}) ⇒ \texttt{fx1 div0 fx2, fx1 mod0 fx2}; two return values
\end{verbatim}

(fx+/carry \texttt{fx1 \texttt{fx2 \texttt{fx3}}}) procedure
Returns the two fixnum results of the following computation:

\begin{verbatim}
(let* ((s (+ \texttt{fx1 \texttt{fx2 \texttt{fx3}}}))
       (s0 (modx s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width))))
       (values s0 s1))
\end{verbatim}

(fx-/carry \texttt{fx1 \texttt{fx2 \texttt{fx3}}}) procedure
Returns the two fixnum results of the following computation:

\begin{verbatim}
(let* ((d (- \texttt{fx1 \texttt{fx2 \texttt{fx3}}}))
       (d0 (modx d (expt 2 (fixnum-width))))
       (d1 (div0 d (expt 2 (fixnum-width))))
       (values d0 d1))
\end{verbatim}
(fx/carry fx1 fx2 fx3)
procedure
Returns the two fixnum results of the following computation:

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

(fxnot fx)
procedure
Returns the unique fixnum that is congruent mod 2^w to the one’s-complement of fx.

(fxand fx1 . . .)
(fxior fx1 . . .)
(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 that is the bit-wise “if” of the two’s complement representations of its arguments, i.e. for each bit, if it is 1 in fx1, the corresponding bit in fx2 becomes the value of the corresponding bit in the result, and if it is 0, the corresponding bit in fx3 becomes the corresponding bit in the value of the result. This is the fixnum result of the following computation:

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

(fxbit-count fx)
procedure
If fx is non-negative, this procedure returns the number of 1 bits in the two’s complement representation of fx. Otherwise it returns the result of the following computation:

(fxnot (fxbit-count (fxnot fx)))

(fxlength fx)
procedure
Returns the number of bits needed to represent fx if it is positive, and the number of bits needed to represent (fxnot fx) if it is negative, which is the fixnum result of the following computation:

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

(fxfirst-bit-set fx)
procedure
Returns the index of the least significant 1 bit in the two’s complement representation of fx. If fx is 0, then −1 is returned.

(fxfirst-bit-set 0) ⇒ -1
(fxfirst-bit-set 1) ⇒ 0
(fxfirst-bit-set -4) ⇒ 2

(fxbit-set? fx1 fx2)
procedure
Fx2 must be non-negative. The fxbit-set? procedure returns #t if the fx2th bit is 1 in the two’s complement representation of fx1, and #f otherwise. This is the result of the following computation:

(if (fx>=? fx2 (fx- (fx-fixnum-width) 1))
  (fxnegative? fx1)
  (not
   (fxzero?
    (fxand fx1
       (fxarithmetic-shift-left 1 fx2))))))

(fxcopy-bit fx1 fx2 fx3)
procedure
Fx2 must be non-negative and less than w − 1. Fx3 must be 0 or 1. The fxcopy-bit procedure returns the result of replacing the fx2th bit of fx1 by fx3, which is the result of the following computation:

(let* (((mask (fxarithmetic-shift-left 1 fx2))
         (fxif mask
            (fxarithmetic-shift-left fx3 fx2)
            fx1))

(fxcopy-bit-field fx1 fx2 fx3)
procedure
Fx2 and fx3 must be non-negative and less than w. Moreover, fx2 must be less than or equal to fx3. The fxcopy-bit-field procedure returns the number represented by the bits at the positions from fx2 (inclusive) to fx3 (exclusive), which is the fixnum result of the following computation:

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

(fxcopy-bit-field fx1 fx2 fx3 fx4)
procedure
Fx2 and fx3 must be non-negative and less than w. Moreover, fx2 must be less than or equal to fx3. The fxcopy-bit-field procedure returns the result of replacing in fx1 the bits at positions from fx2 (inclusive) to fx3 (exclusive) by the bits in fx4 from position 0 (inclusive) to position fx3 − fx2 (exclusive), which is the fixnum result of the following computation:
significant bits, which is the result of the following computation:

\[
\text{fxrotate-bit-field} \quad \text{fx}
\]

The absolute value of \( f2 \) must be less than \( w \). If \( \text{(floor} (* f1 (\text{expt} \ 2 f2)) \) is a fixnum, then that fixnum is returned. Otherwise an exception with condition type \&implementation-restriction is raised.

\[
\text{fxarithmetic-shift-left} \quad \text{fx}
\]

\[
\text{fxarithmetic-shift-right} \quad \text{fx}
\]

These procedures return the result of cyclically permuting in \( f1 \) the bits at positions from \( f2 \) (inclusive) to \( f3 \) (exclusive) by \( f4 \) bits towards the more significant bits, which is the result of the following computation:

\[
\text{let*} \ ((n \ f1) \ (start f2) \ (end f3) \ (from f4) \ (mask1 (\text{fxarithmetic-shift-left} \ -1 \ start)) \ (mask2 (\text{fxnot} \ (\text{fxarithmetic-shift-left} \ -1 \ end))) \ (mask3 (\text{fxnot} \ (\text{fxarithmetic-shift-left} \ -1 \ (- \ end \ start)))) \ (\text{fxif} \ mask \ (\text{fxarithmetic-shift-left} \ (\text{fxand} \ from \ mask3) \ start) \ to)) \ (\text{fxcopy-bit-field} \ #b0000001 \ 2 \ 5 \ #b1111000) \ \Rightarrow \ 1 \ (\text{fxcopy-bit-field} \ #b0000001 \ 2 \ 5 \ #b0001111) \ \Rightarrow \ 29 \ (\text{fxcopy-bit-field} \ #b0001111 \ 2 \ 5 \ #b0001111) \ \Rightarrow \ 31 \ (\text{fxarithmetic-shift-left} \ f1 \ f2) \ \text{procedure} \ Fx2 \ \text{must be non-negative, and less than} \ w. \ The \ \text{fxarithmetic-shift-left} \ \text{procedure behaves the same as} \ \text{fxarithmetic-shift}, \ \text{and} \ (\text{fxarithmetic-shift-right} \ f1 \ f2) \ \text{behaves the same as} \ (\text{fxarithmetic-shift} \ f1 \ (\text{fx} \ f2)). \ (\text{fxrotate-bit-field} \ f1 \ f2 \ f3 \ f4) \ \text{procedure} \ Fx2, \ f3, \ \text{and} \ f4 \ \text{must be non-negative and less than} \ w. \ Fx2 \ \text{must be less than or equal to} \ f3, \ Fx4 \ \text{must be less than or equal to} \ \text{the difference between} \ f3 \ \text{and} \ f2. \ \text{The} \ \text{fxrotate-bit-field} \ \text{procedure returns the result of cyclically permuting in} \ f1 \ \text{the bits at positions from} \ f2 \ \text{(inclusive)} \ \text{to} \ f3 \ \text{(exclusive)} \ \text{by} \ f4 \ \text{bits towards the more significant bits, which is the result of the following computation:}

\[
\text{let*} \ ((n \ f1) \ (start f2) \ (end f3) \ (count f4) \ (width (\text{fx} \ - \ end \ start))) \ (\text{fxcopy-bit-field} \ n \ start \ end) \ (\text{fxior}) \ (\text{fxarithmetic-shift-left} \ (\text{fxbit-field} \ n \ start \ (\text{fx} \ - \ end \ count))) \ (\text{fxreverse-bit-field} \ f1 \ f2 \ f3) \ \text{procedure} \ Fx2 \ \text{and} \ f3 \ \text{must be non-negative and less than} \ w. \ Moreover, f2 \ \text{must be less than or equal to} \ f3. \ \text{The} \ \text{fxreverse-bit-field} \ \text{procedure returns the fixnum obtained from} \ f3 \ \text{by reversing the order of the bits at positions from} \ f2 \ \text{(inclusive)} \ \text{to} \ f3 \ \text{(exclusive)}. \ (\text{fxreverse-bit-field} \ #b1010010 \ 1 \ 4) \ \Rightarrow \ 88 \ ; \ #b1011000

11.3. Flonums

This section describes the (rnrs arithmetic flonums (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.

(flnot \ fl) \ \text{procedure} \ Returns \ #t \ \text{if} \ obj \ \text{is a flonum,} \ #f \ \text{otherwise.}

(flreal->flonum \ x) \ \text{procedure} \ Returns the best flonum representation of \ x.

(fl=?) \ \text{procedure} \ Arguments that must be integer-valued flonums, i.e., flonums for which the integer-valued? predicate returns true.

(fl=<?) \ \text{procedure} \ These procedures return \ #t \ if their arguments are (respectively): equal, monotonically increasing, monotonically nondecreasing, monotonically decreasing, or monotonically nonincreasing, \ #f \ otherwise. These predicates must be transitive.
(flinteger? fl) procedure
(flover? fl) procedure
(flpositive? fl) procedure
(flnegative? fl) procedure
(flodd? ifl) procedure
(fleven? ifl) procedure
(flfinite? fl) procedure
(flfinite? 5.0) procedure
(flfinite? 5.0) procedure
(flfinite? +inf.0) procedure

Note: (flnegative? -0.0) must return #f, else it would lose the correspondence with (flc? -0.0 0.0), which is #f according to IEEE 754 [7].

(flmax fl1 fl2 ...) procedure
(flmin fl1 fl2 ...) procedure

These procedures return the maximum or minimum of their arguments. They always return a NaN when one or more of the arguments is a NaN.

(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 reasonably well-defined by the IEEE standards.)

(fl+ +inf.0 -inf.0) => +nan.0
(fl+ +nan.0 fl) => +nan.0
(fl* +nan.0 fl) => +nan.0

 asociating 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.)

(fl- +inf.0 +inf.0) => +nan.0

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

(fl/ 1.0 0.0) => +inf.0
(fl/ -1.0 0.0) => -inf.0
(fl/ 0.0 0.0) => +nan.0

(flabs fl) procedure
Returns the absolute value of fl.

(fldiv-and-mod fl1 fl2) procedure
(fldiv fl1 fl2) procedure
(flmod fl1 fl2) procedure
(fldiv0-and-mod0 fl1 fl2) procedure
(fldiv0 fl1 fl2) procedure
(flmod0 fl1 fl2) procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section 11.7.3. In the cases where the mathematical requirements in section 11.7.3 cannot be satisfied by any number object, either an exception is raised with condition type implementation-restriction, or unspecified flonums (one for fldiv flmod, fldiv0 and flmod0, two for fldiv-and-mod and fldiv0-and-mod0) are returned.

(fldiv fl1 fl2) => fl1 div fl2
(flmod fl1 fl2) => fl1 mod fl2
(fldiv-and-mod fl1 fl2) => fl1 div fl2, fl1 mod fl2 ; two return values
(fldiv0 fl1 fl2) => fl1 div0 fl2
(flmod0 fl1 fl2) => fl1 mod0 fl2 ; two return values
(fldiv0-and-mod0 fl1 fl2) => fl1 div0 fl2, fl1 mod0 fl2 ; two return values

(flnumerator fl) procedure
(fldenominator fl) procedure

These procedures return the numerator or denominator of fl as a flonum; the result is computed as if fl was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0.0 is defined to be 1.0.
Implementations should implement following behavior:

\[(\text{flnumerator} \, -0.0) \quad \Rightarrow \quad -0.0\]

These procedures return integral flonums for flonum arguments that are not infinities or NaNs. For such arguments, \(\text{flfloor}\) returns the largest integral flonum not larger than \(fl\). The \(\text{flceiling}\) procedure returns the smallest integral flonum not smaller than \(fl\). The \(\text{fltruncate}\) procedure returns the integral flonum closest to \(fl\) whose absolute value is not larger than the absolute value of \(fl\). The \(\text{flround}\) 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:

\[(\text{flfloor} \, +\infty.0) \quad \Rightarrow \quad +\infty.0\]
\[(\text{flceiling} \, -\infty.0) \quad \Rightarrow \quad -\infty.0\]
\[(\text{fltruncate} \, +\infty.0) \quad \Rightarrow \quad +\infty.0\]

These procedures compute the usual transcendental functions. The \(\text{flexp}\) procedure computes the base-\(e\) exponential of \(fl\). The \(\text{fllog}\) procedure with a single argument computes the natural logarithm of \(fl\) (not the base ten logarithm); \(\text{fllog} \, fl_1 \, fl_2\) computes the base-\(fl_2\) logarithm of \(fl_1\). The \(\text{flasin}, \text{flacos}, \text{and} \text{flatan}\) procedures compute arcsine, arccosine, and arctangent, respectively. \(\text{flatan} \, fl_1 \, fl_2\) computes the arc tangent of \(fl_1 / fl_2\).

See report section 11.7.3 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 unspecified flonum.
These types describe that a program has executed an arithmetic operations that is specified to return an infinity or a NaN, respectively, on a Scheme implementation that is not able to represent the infinity or NaN. (See report section 11.7.2)

\(\text{(fixnum->flonum } fx)\) procedure

Returns a flonum that is numerically closest to \(fx\).

Note: The result of this procedure may not be numerically equal to \(fx\), because the fixnum precision may be greater than the flonum precision.

### 11.4. 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 \(ei\), \(e1\), \(e2\), etc., as parameter names that must be exact integer objects.

\(\text{(bitwise-not } ei)\) procedure

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

\(\text{(bitwise-and } ei_1 \ldots)\) procedure
\(\text{(bitwise-ior } ei_1 \ldots)\) procedure
\(\text{(bitwise-xor } ei_1 \ldots)\) 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.

\(\text{(bitwise-if } ei_1 \ e1_2 \ e1_3)\) procedure

Returns the exact integer object that is the bit-wise “if” of the two’s complement representations of its arguments, i.e. for each bit, if it is 1 in \(e1_1\), the corresponding bit in \(e1_2\) becomes the value of the corresponding bit in the result, and if it is 0, the corresponding bit in \(e1_3\) becomes the corresponding bit in the value of the result. This is the result of the following computation:

\(\text{(bitwise-ior (bitwise-and } ei_1 \ e1_2)\)}
\((\text{bitwise-and (bitwise-not } ei_1 \ e1_3)\))

\(\text{(bitwise-bit-count } ei)\) procedure

If \(ei\) is non-negative, this procedure returns the number of 1 bits in the two’s complement representation of \(ei\). Otherwise it returns the result of the following computation:

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

\(\text{(bitwise-length } ei)\) procedure

Returns the number of bits needed to represent \(ei\) if it is positive, and the number of bits needed to represent \(\text{(bitwise-not } ei)\) if it is negative, which is the exact integer object that is the result of the following computation:

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

\(\text{(bitwise-first-bit-set } ei)\) 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.

\(\text{(bitwise-first-bit-set } 0) \quad \Rightarrow \quad -1\)
\(\text{(bitwise-first-bit-set } 1) \quad \Rightarrow \quad 0\)
\(\text{(bitwise-first-bit-set } -4) \quad \Rightarrow \quad 2\)

\(\text{(bitwise-bit-set? } ei_1 \ e1_2\)) procedure

\(\text{Ei}\) must be non-negative. The \(\text{(bitwise-bit-set? } ei_1 \ e1_2\)) procedure returns \#t if the \(\text{e1}\)th bit is 1 in the two’s complement representation of \(\text{e1}\), and \#f otherwise. This is the result of the following computation:

\(\text{(not (zero?}
\text{(bitwise-and}
\text{(bitwise-arithmetic-shift-left 1 } ei_2)
\text{ei_1))))\)

\(\text{(bitwise-copy-bit } ei_1 \ e1_2 \ e1_3)\) procedure

\(\text{Ei}\) must be non-negative, and \(e1_3\) must be either 0 or 1. The \(\text{(bitwise-copy-bit } ei_1 \ e1_2 \ e1_3)\) procedure returns the result of replacing the \(\text{e1}\)th bit of \(\text{ei}\) by \(\text{e1}\), which is the result of the following computation:

\(\text{(let* ((mask (bitwise-arithmetic-shift-left 1 } ei_2))}
\text{(bitwise-if mask}
\text{(bitwise-arithmetic-shift-left } ei_3 \ ei_2)
\text{ei_1))))\)

\(\text{(bitwise-bit-field } ei_1 \ e1_2 \ e1_3)\) procedure

\(\text{Ei}\) and \(\text{e1}\) must be non-negative, and \(\text{e1}\) must be less than or equal to \(\text{e1}\). The \(\text{(bitwise-bit-field } ei_1 \ e1_2 \ e1_3)\) procedure returns the number represented by the bits at the positions from \(\text{e1}\) (inclusive) to \(\text{e1}\) (exclusive), which is 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 replacing in ei1 the bits at positions from ei2 (inclusive) to ei3 (exclusive) by the bits in ei4 from position 0 (inclusive) to position ei3 − ei2 (exclusive), which is the result of the following computation:

(let* ((to ei1)
  (start ei2)
  (end ei3)
  (from ei4)
  (mask1 (bitwise-arithmetic-shift-left -1 start))
  (mask2 (bitwise-not
    (bitwise-arithmetic-shift-left -1 end)))
  (mask (bitwise-and mask1 mask2)))
(bitwise-if mask
  (bitwise-arithmetic-shift-left from
   start)
  to))

(bitwise-arithmetic-shift ei1 ei2) procedure

Returns the result of the following computation:

(floor (* ei1 (expt 2 ei2)))

Examples:

(bitwise-arithmetic-shift -6 -1)  ⇒ -3
(bitwise-arithmetic-shift -5 -1)  ⇒ -3
(bitwise-arithmetic-shift -4 -1)  ⇒ -2
(bitwise-arithmetic-shift -3 -1)  ⇒ -2
(bitwise-arithmetic-shift -2 -1)  ⇒ -1
(bitwise-arithmetic-shift -1 -1)  ⇒ -1

(bitwise-arithmetic-shift-left ei1 ei2) procedure

Ei2 must be non-negative. The bitwise-arithmetic-shift-left procedure returns the same result as bitwise-arithmetic-shift, and

(bitwise-arithmetic-shift-right ei1 ei2)
returns the same result as

(bitwise-arithmetic-shift ei1 (- ei2)).

(bitwise-rotate-bit-field ei1 ei2 ei3 ei4) procedure

Ei2, ei3, ei4 must be non-negative, ei2 must be less than or equal to ei3. The bitwise-rotate-bit-field procedure returns the result of cyclically permuting in ei1 the bits at positions from ei2 (inclusive) to ei3 (exclusive) by ei4 bits towards the more significant bits, which is the result of the following computation:

(let* ((n ei1)
  (start ei2)
  (end ei3)
  (count ei4)
  (width (- end start)))
(if (positive? width)
  (let* ((count (mod count width))
    (field0 (bitwise-bit-field n start end))
    (field1 (bitwise-arithmetic-shift-left
      field0 count))
    (field2 (bitwise-arithmetic-shift-right
      field0 (- width count)))
    (field (bitwise-ior field1 field2)))
  (bitwise-copy-bit-field n start end field))

(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 order of the bits at positions from ei2 (inclusive) to ei3 (exclusive).

(bitwise-reverse-bit-field #b1010010 1 4)
⇒ 88 ; #b1011000

12. syntax-case

The (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 [2] 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 [3]. Thus, the hygiene condition can be restated as follows:

A binding for an identifier introduced into the output of a transformer call from the expander must capture only references to the identifier introduced into the output of the same transformer call. A reference to an identifier introduced into the output of a transformer refers to the closest enclosing binding for the introduced identifier or, if it appears outside of any enclosing binding for the introduced identifier, the closest enclosing lexical binding where the identifier appears (within a syntax (template)) inside the transformer body or one of the helpers it calls.

Explicit captures are handled via datum->syntax; see section 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 lambda expression, the expander might map each bound identifier to a representation of the formal parameter in the output of the expander. For a let-syntax form, the expander might map each bound identifier to the associated transformer.) These substitutions are applied to the portions of the input form in which the binding is supposed to be visible.

Marks and substitutions together form a wrap that is layered on the form being processed by the expander and pushed down toward the leaves as necessary. A wrapped form is referred to as a wrapped syntax object. Ultimately, the wrap may rest on a leaf that represents an identifier, in which case the wrapped syntax object is also referred to as an identifier. An identifier contains a name along with the wrap. (Names are typically represented by symbols.)

When a substitution is created to map an identifier to an expand-time value, the substitution records the name of the identifier and the set of marks that have been applied to that identifier, along with the associated expand-time value. The expander resolves identifier references by looking for the latest matching substitution to be applied to the identifier, i.e., the outermost substitution in the wrap whose name and marks match the name and marks recorded in the substitution. The name matches if it is the same name (if using symbols, then by eq?), and the marks match if the marks recorded with the substitution are the same as those that appear below the substitution in the wrap, i.e., those that were applied before the substitution. Marks applied after a substitution, i.e., appear over the substitution in the wrap, are not relevant and are ignored.

An algebra that defines how marks and substitutions work more precisely is given in section 2.4 of Oscar Waddell’s PhD thesis [13].

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.

A syntax object may be wrapped, as described in section 12.1. It may also be unwrapped, fully or partially, i.e., consist of list and vector structure with wrapped syntax objects or nonsymbol values at the leaves. More formally, a syntax object is:

- a pair of syntax objects,
- a vector of syntax objects,
• a nonpair, nonvector, nonsymbol value, or
• a wrapped syntax object.

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

Syntax objects representing identifiers are always wrapped and are distinct from other types of values. Wrapped syntax objects that are not identifiers may or may not be distinct from other types of values.

12.3. Transformers

In define-syntax (report section 11.2.2), let-syntax, and letrec-syntax forms (report section 11.18), a binding for a syntactic keyword is an expression that evaluates to a transformer.

A transformer is a transformation procedure or a variable transformer. A transformation procedure is a procedure that must accept one argument, a wrapped syntax object (section 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 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 variable transformers, an exception with condition type #syntax is raised if the keyword appears on the left-hand side of a set! expression.

(make-variable-transformer proc) procedure
Proc 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 set! expression, an exception is not raised. Instead, proc is called with a wrapped syntax object representing the entire set! expression as its argument, and its return value replaces the entire set! expression.

Implementation responsibilities: The implementation must check the restrictions on proc only to the extent performed by applying it as described. An implementation may check whether proc is an appropriate argument before applying it.

12.4. Parsing input and producing output

Transformers can destructure their input with syntax-case and rebuild their output with syntax.

(syntax-case (expression) ((literal) ...)) syntax
  (syntax-case clause) ...

... auxiliary syntax
... auxiliary syntax

Syntax: Each (literal) must be an identifier. Each (syntax-case-clause) must take one of the following two forms.

  ((pattern) (output expression))
  ((pattern) (fender) (output expression))

(Fender) and (output expression) must be (expression)/s.

A (pattern) is an identifier, constant, or one of the following.

  ((pattern) ...) (pattern) ... . (pattern)
  ((pattern) ... (pattern) (ellipsis) (pattern) ...)
  ((pattern) ... (pattern) (ellipsis) (pattern) ... . (pattern))
  #((pattern) ...)
  #((pattern) ... (pattern) (ellipsis) (pattern) ...)

An (ellipsis) is the identifier “...” (three periods).

An identifier appearing within a (pattern) may be an underscore (_), a literal identifier listed in the list of literals ((literal) ...), or an ellipsis (...). All other identifiers appearing within a (pattern) are pattern variables. It is a syntax violation if an ellipsis or underscore appears in ((literal) ...).

_and ... are the same as in the (rnrs base (6)) library.

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 [12.3]).
• P is of the form (P₁ ... Pₙ) and F is a list of n elements that match P₁ through Pₙ.
• P is of the form (P₁ ... Pₙ . P₂) and F is a list or improper list of n or more elements whose first n elements match P₁ through Pₙ and whose nth cdr matches P₂.
• P is of the form (P₁ ... Pₖ Pₑ (ellipsis) Pₘ₊₁ ... Pₙ), where (ellipsis) is the identifier ... and F is a proper list of n elements whose first k elements match P₁ through Pₖ, whose next m − k elements each match Pₑ, and whose remaining n − m elements match Pₘ₊₁ through Pₙ.
• P is of the form (P₁ ... Pₖ Pₑ (ellipsis) Pₘ₊₁ ... Pₙ . P₂), where (ellipsis) is the identifier ... and F is a list or improper list of n elements whose first k elements match P₁ through Pₖ, whose next m − k elements each match Pₑ, whose next n − m elements match Pₘ₊₁ through Pₙ, and whose nth and final cdr matches P₂.
• P is of the form #(P₁ ... Pₙ) and F is a vector of n elements that match P₁ through Pₙ.
• P is of the form #(P₁ ... Pₖ Pₑ (ellipsis) Pₘ₊₁ ... Pₙ), where (ellipsis) is the identifier ... and F is a vector of n or more elements whose first k elements match P₁ through Pₖ, whose next m − k elements each match Pₑ, and whose remaining n − m elements match Pₘ₊₁ through Pₙ.
• P is a pattern datum (any nonlist, nonvector, non-symbol datum) and F is equal to P in the sense of equal? procedure.

Semantics: A syntax-case expression first evaluates (expression). It then attempts to match the (pattern) from the first (syntax-case 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 (syntax-case 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 (syntax-case 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.

If the syntax-case form is in tail context, the (output expression)s are also in tail position.

(syntax (template))
Note: #\(\) (template) is equivalent to (syntax (template)).

A syntax expression is similar to a quote expression except that (1) the values of pattern variables appearing within (template) are inserted into (template), (2) contextual information associated both with the input and with the template is retained in the output to support lexical scoping, and (3) the value of a syntax expression is a syntax object.

A (template) is a pattern variable, an identifier that is not a pattern variable, a pattern datum, or one of the following.

\((\text{subtemplate}) \ldots\)  
\((\text{subtemplate}) \ldots . (\text{template})\)  
#(\text{subtemplate}) \ldots\)

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 \((\langle \text{ellipsis} \rangle \langle \text{template} \rangle)\) is identical to \langle \text{template} \rangle, except that ellipses within the template have no special meaning. That is, any ellipses contained within \langle \text{template} \rangle are treated as ordinary identifiers. In particular, the template \((\ldots \ldots)\) produces a single ellipsis. This allows macro uses to expand into forms containing ellipses.

The output produced by \texttt{syntax} is wrapped or unwrapped according to the following rules.

- the copy of \((\langle t_1 \rangle . \langle t_2 \rangle)\) is a pair if \langle t_1 \rangle or \langle t_2 \rangle contain any pattern variables,
- the copy of \((\langle t \rangle \langle \text{ellipsis} \rangle)\) is a list if \langle t \rangle contains any pattern variables,
- the copy of \#(\langle t_1 \rangle \ldots \langle t_n \rangle)\) is a vector if any of \langle t_1 \rangle, \ldots, \langle t_n \rangle contain any pattern variables, and
- the copy of any portion of \langle t \rangle 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 \texttt{or} illustrate \texttt{syntax-case} and \texttt{syntax}. The second is equivalent to the first but uses the \texttt{#'} prefix instead of the full \texttt{syntax} form.

\begin{verbatim}
(define-syntax or
 (lambda (x)
 (syntax-case x ()
 [((.) (syntax #f)]
 [([e] (syntax e)]
 [([e1 e2 e3 ...]
 (syntax (let ([t e1]
 (if t t (or e2 e3 ...))))))]))

(define-syntax or
 (lambda (x)
 (syntax-case x ()
 [((.) #'#f]
 [([e] #'e]
 [([e1 e2 e3 ...]
 #'(let ([t e1]
 (if t t (or e2 e3 ...))))))))

The examples below define identifier macros, macro uses supporting keyword references that do not necessarily appear in the first position of a list-structured form. The second example uses \texttt{make-variable-transformer} to handle the case where the keyword appears on the left-hand side of a \texttt{set!} expression.

\begin{verbatim}
(define p (cons 4 5))
(define-syntax p.car
 (lambda (x)
 (syntax-case x ()
 [((.) rest) #'((car p) . rest)]
 [([e1 e2 e3 ...]
 #'(let ([t e1]
 (if t t (or e2 e3 ...))))))]))

(set! p.car 15)
⇒ 15
p
⇒ (15 . 5)
\end{verbatim}

\section{12.5. Identifier predicates}

\begin{verbatim}
(identifier? obj)
\end{verbatim}

procedure

Returns \#t if \texttt{obj} is an identifier, i.e., a syntax object representing an identifier, and \#f otherwise. The \texttt{identifier?} procedure is often used within a fender to verify that certain subforms of an input form are identifiers, as in the definition of \texttt{rec}, which creates self-contained recursive objects, below.

\begin{verbatim}
(define-syntax rec
 (lambda (x)
 (syntax-case x ()
 [((x e) (identifier? #'x)
 #'(letrec ([x e]) x)])
 [([e1 e2 e3 ...]
 (if t t (or e2 e3 ...))))]))

(map rec fact
 (lambda (n)
 (if (= n 0)
 1
 (+ n (fact (- n 1)))))))

(rec 5 (lambda (x) x))
⇒ \&syntax exception
\end{verbatim}

The procedures \texttt{bound-identifier=?} and \texttt{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.

\begin{verbatim}
(bound-identifier=? \texttt{id}_1 \texttt{id}_2)
\end{verbatim}

procedure

\texttt{Id}_1 and \texttt{Id}_2 must be identifiers. The procedure \texttt{bound-identifier=?} returns \#t 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, and #f otherwise. 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.

The syntax-case and syntax-rules forms internally use free-identifier=? to compare identifiers listed in the literals list against input identifiers.

(let ([fred 17])
  (define-syntax a
    (lambda (x)
      (syntax-case x ()
        [(id) #'(b id fred)])))
  (define-syntax b
    (lambda (x)
      (syntax-case x ()
        [(id1 id2)
          #'(list
              (free-identifier=? #'id1 #'id2)
              (bound-identifier=? #'id1 #'id2))])))
(a fred) => (#t #f)

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)))))))))

With either definition of case, else is not recognized as an auxiliary keyword if an enclosing lexical binding for else exists. For example,

(let ([else #f])
  (case 0 [else (write "oops")])
  => &syntax exception

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

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=?.

(define-syntax case
  (lambda (x)
    (syntax-case x ()
      [(e0 [(ka ...) e1a e2a ...] [(kb ...) e1b e2b ...])
        #'(let ([t e0])
            (cond
              [(memv t '(ka ...)) e1a e2a ...]
              [(memv t '(kb ...)) e1b e2b ...]
              ...)))
      [(e0 [(k ...) e1 e2 ...] ...
          [else-key else-e1 else-e2 ...])
        #'(let ([t e0])
            (cond
              [(memv t '(k ...)) e1 e2 ...]
              ...]
              [else else-e1 else-e2 ...])])

With either definition of case, else is not recognized as an auxiliary keyword if an enclosing lexical binding for else exists. For example,

(let ([else #f])
  (case 0 [else (write "oops")])
  => &syntax exception

since else is bound lexically and is therefore not the same else that appears in the definition of case.
12.6. Syntax-object and datum conversions

(syntax->datum syntax-object)  procedure

Strips all syntactic information from a syntax object and
returns the corresponding Scheme datum.

Identifiers stripped in this manner are converted to their
symbolic names, which can then be compared with eq?.
Thus, a predicate symbolic-identifier=? might be defined as follows.

(define symbolic-identifier=?
  (lambda (x y)
    (eq? (syntax->datum x)
         (syntax->datum y))))

(datum->syntax template-id datum)  procedure

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

The datum->syntax procedure allows a transformer to
“bend” lexical scoping rules by creating implicit identi-
fiers that behave as if they were present in the input form, thus permitting the definition of macros that introduce vis-
ible bindings for or references to identifiers that do not
appear explicitly in the input form. For example, the fol-
lowing defines a loop expression that uses this controlled
form of identifier capture to bind the variable break to
an escape procedure within the loop body. (The derived
with-syntax form is like let but binds pattern variables—
see section 12.8.)

(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))))])))

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

Were loop to be defined as

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

the variable break would not be visible in e ....

The datum argument datum may also represent an arbitrary Scheme form, as demonstrated by the following defini-
tion of include.

(define-syntax include
  (lambda (x)
    (define read-file
      (lambda (fn k)
        (let ([p (open-file-input-port fn
                    (file-options)
                    (buffer-mode block)
                    (native-transcoder))])
          (let f ([x (get-datum p)])
            (if (eof-object? x)
                (begin (close-port p) '())
                (cons (datum->syntax k x)
                      (f (get-datum p)))))
        (let ([fn (syntax->datum #'filename)])
          (with-syntax ([[(exp ...) (read-file fn #'#k)]]
                       #'(begin exp ...)))))

(include "filename") expands into a begin expres-
sion containing the forms found in the file named by
"filename". For example, if the file flib.ss contains
(define f (lambda (x) (g (* x x))))
and the file glib.ss contains (define g (lambda (x) (+ x x)));
the expression

(let ()
  (include "flib.ss")
  (include "glib.ss")
  (f 5))

evaluates to 50.

The definition of include uses datum->syntax to convert
the objects read from the file into syntax objects in the
proper lexical context, so that identifier references and
definitions within those expressions are scoped where the
include form appears.

Using datum->syntax, it is even possible to break hygiene
entirely and write macros in the style of old Lisp macros.
The lisp-transformer procedure defined below creates a
transformer that converts its input into a datum, calls the
programmer's procedure on this datum, and converts the
result back into a syntax object scoped where the original
macro use appeared.

(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))))])))

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 scoped where the original
macro use appeared.
(define lisp-transformer
(lambda (p)
  (lambda (x)
    (syntax-case x ()
      [(kw . rest)
        (datum->syntax #'kw
          (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 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 ...⟩
      (with-syntax (((t ...) (generate-temporaries #'(i ...)))
        #'(let ((i <undefined>) ...) (let ((t e) ...) (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 can be defined in terms of the forms and procedures described in earlier sections of this chapter.

(with-syntax ((⟨pattern⟩ ⟨expression⟩) ...) ⟨body⟩)

The 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 ()
    [(⟨(i e) ...) 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 ...) e1 e2 ...⟩
      (let f ([⟨c1 #'c1⟩ ⟨c2* #'(c2 ...)⟩]
        (syntax-case c2* ()
          [(⟨c1 c2 ...) e1 e2 ...⟩
            (with-syntax ([rest (f #'c2 #'(c3 ...)])
              (syntax-case c1 (=>)
                [(⟨e0⟩ #'l) (let ((i e) ...) (set! i t) ...
                  (let () e1 e2 ...)))
                [(⟨e0⟩ #e0) #'e0]
                [(⟨e0⟩ => e1) #'(let ((t e0) (if t (e1 t))))
                  [(⟨e0 e1 e2 ...) #'(if e0 (begin e1 e2 ...) rest))]]
                ))
            ]))
    )))

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 11.17). 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 quasiquote expressions.

A quasiquote expression may be nested, with each quasiquote introducing a new level of syntax quotation and each unsyntax or unsyntax-splicing taking away a level of quotation. An expression nested within \( n \) quasiquote expressions must be within \( n \) unsyntax or unsyntax-splicing expressions to be evaluated.

As noted in report section \( \text{\textsection}4.3.5 \) \( \text{\textsection}11.19 \) may be defined in terms of syntax-case, syntax, and make-variable-transformer as follows.

\[
\begin{align*}
\text{(define-syntax identifier-syntax} \& \text{lambda} (x) \\
\text{(syntax-case x (set!)}) \& \text{lambda} (x) \\
\text{(#\{lambda \{x\} \}} \& \text{(syntax-case x (set!}} \& \text{)} \& \text{procedure} \\
\text{(syntax-violation \} who message form} \}} \& \text{procedure} \\
\text{(syntax-violation \} message form subform} \}} \& \text{procedure} \\
\text{Who must be \#f or a string or a symbol. Message must be a string. Form must be a syntax object or a datum value. Subform must be a syntax object or a datum value. The syntax-violation procedure raises an exception, reporting a syntax violation. Who should describe the macro transformer that detected the exception. The message argument should describe the violation. Form should be the erroneous source syntax object or a datum value representing a form. The optional subform argument should be a syntax object or datum value representing a form that more precisely locates the violation.} \\
\text{If who is \#f, syntax-violation attempts to infer an appropriate value for the condition object (see below) as follows: When form is either an identifier or a list-structured syntax object containing an identifier as its first element, then the inferred value is the identifier's symbol. Otherwise, no value for who is provided as part of the condition object.} \\
\text{The condition object provided with the exception (see chapter \#7 has the following condition types:} \\
\end{align*}
\]
• If `who` is not `#f` or can be inferred, the condition has condition type `&who`, with `who` as the value of its field. Otherwise, the condition does not have condition type `&who`.

• The condition has condition type `&message`, with `message` as the value of its field.

• The condition has condition type `&syntax` with `form` and `subform` as the value of its fields. If `subform` is not provided, the value of the subform field is `#f`.

13. Hashtables

The `(rners hashtables)` 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 11.5) 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.

`(makeHashtable hash-function equiv)` procedure
`(makeHashtable 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-eqv` 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.

13.2. Procedures

`(hashtable? obj)` procedure
Returns `#t` if `obj` 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:

(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
  ; entries may be in different order

13.3. Inspection

(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.

13.4. 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 enums (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.

(match-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.

(list (enum-set-universe enum-set)) procedure
Returns the set of all symbols that comprise the universe of its argument, as an enumeration set.

(list (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 symbol 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)
      (cond
       ((memq x symbols)
        => (lambda (probe)
            (- cardinality (length probe))))
       (else #f))))

(match-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.

(match-set->list enum-set) procedure
Returns a list of the symbols that belong to its argument, in the canonical order of the universe of enum-set.

(let* ((e (make-enumeration '(red green blue)))
        (c (enum-set-constructor e))
        (enum-set->list (c '(blue red))))
⇒ (red blue)

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

(match-set-subset? enum-set1 enum-set2) procedure
The match-set-subset? procedure returns #t if the universe of enum-set1 is a subset of the universe of enum-set2 (considered as sets of symbols) and every element of enum-set1 is a member of enum-set2. It returns #f otherwise.

(match-set=? enum-set1 enum-set2) procedure
The match-set=? procedure returns #t if enum-set1 is a subset of enum-set2 and vice versa, as determined by the match-set-subset? procedure. This implies that the universes of the two sets are equal as sets of symbols, but does not imply that they are equal as enumeration types. Otherwise, #f is returned.

(let* ((e (make-enumeration '(red green blue)))
        (c (enum-set-constructor e)))
  (list
   (enum-set-member? 'blue (c '(blue red)))
   (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=? (c '(red blue)) (c '(red)))
   (enum-set=? (c '(red blue)) (c '(blue red))))
⇒ (#t #f #t #t)

(match-set-union enum-set1 enum-set2) procedure
(match-set-intersection enum-set1 enum-set2) procedure
(match-set-difference enum-set1 enum-set2) procedure

Enum-set1 and enum-set2 must be enumeration sets that have the same enumeration type.
The `enum-set-union` procedure returns the union of `enum-set_1` and `enum-set_2`. The `enum-set-intersection` procedure returns the intersection of `enum-set_1` and `enum-set_2`. The `enum-set-difference` procedure returns the difference of `enum-set_1` and `enum-set_2`.

```scheme
(let* ((e (make-enumeration '(red green blue)))
        (c (enum-set-constructor e)))
  (list (enum-set-union (c '(blue)) (c '(red))))
  (enum-set-intersection (c '(red green))
                         (c '(red blue)))
  (enum-set-difference (c '(red green))
                        (c '(red blue))))
⇒ ((red blue) (red) (green))
```

The `enum-set-complement` procedure returns `enum-set`'s complement with respect to its universe.

```scheme
(let* ((e (make-enumeration '(red green blue)))
        (c (enum-set-constructor e)))
  (enum-set->list (enum-set-complement (c '(red)))))
⇒ (green blue)
```

The `enum-set-projection` procedure projects `enum-set_1` into the universe of `enum-set_2`, dropping any elements of `enum-set_1` that do not belong to the universe of `enum-set_2`. (If `enum-set_1` is a subset of the universe of its second, no elements are dropped, and the injection is returned.) The result has the enumeration type of `enum-set_2`.

```scheme
(let* ((e1 (make-enumeration 'black white purple maroon))
        (e2 (make-enumeration 'black white))))
  (enum-set->list (enum-set-projection e1 e2)))
⇒ (red black)
```

A `define-enumeration` form is a definition and can appear anywhere any other (definition) can appear.

- `(define-enumeration ⟨type-name⟩ syntax ⟨⟨symbol⟩ . . .⟩ ⟨⟨constructor-syntax⟩⟩)`
- `(define-enumeration ⟨⟨type-name⟩ ⟨⟨symbol⟩ . . .⟩⟩ ⟨⟨constructor-syntax⟩⟩)`

The `define-enumeration` form defines an enumeration type and provides two macros for constructing its members and sets of its members.

- `⟨⟨symbol⟩ . . .⟩` checks at macro-expansion time whether the name of `⟨⟨symbol⟩ . . .⟩` is in the universe associated with `⟨⟨type-name⟩ . . .⟩`. If it is, `⟨⟨constructor-syntax⟩ ⟨⟨symbol⟩ . . .⟩⟩` is equivalent to `⟨⟨type-name⟩ ⟨⟨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 `((enum-set-constructor ⟨⟨constructor-syntax⟩⟩) '⟨⟨symbol⟩ . . .⟩)`. Example:

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

- `(color black)` ➞ black
- `(color purpel)` ➞ &syntax exception
- `(enum-set->list (color-set))` ➞ ()
- `(enum-set->list (color-set maroon white))` ➞ (white maroon)

Note: In `⟨⟨type-name⟩ ⟨⟨symbol⟩ . . .⟩⟩` and `⟨⟨constructor-syntax⟩ ⟨⟨symbol⟩ . . .⟩⟩` forms, only the names of the `⟨⟨symbol⟩ . . .⟩`s are significant.

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 7.2

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) procedure
Evaluates expression in the specified environment and returns its value. Expression must be a syntactically valid Scheme expression represented as a datum value, and environment must be an environment, 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 7.1). 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.

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.

(set-cdr! pair obj) procedure
Stores obj in the cdr field of pair. The set-cdr! procedure returns unspecified values.

If an immutable pair is passed to set-car!, an exception with condition type &assertion should be raised.

If an immutable pair is passed to set-cdr!, an exception with condition type &assertion should be raised.

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.


Note: Implementors should 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 (r5rs r5rs (6)) library and provide some functionality of the preceding revision of this report [8] that was omitted from the main part of the current report.

Note: These procedures could be defined in terms of div and mod (see report section 11.7.4) 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 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. The force procedure 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.
(lambda (n)
  (cons n (delay (next (+ n 1))))))

(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail a-stream)))
⇒ 2

Promises are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
              (if (> count x)
                  count
                  (force p))))))

(define x 5)

p
⇒ a promise
(force p)
⇒ 6
p
⇒ a promise, still
(begin (set! x 10)
       (force p))
⇒ 6

Here is a possible implementation of delay and force. Promises are implemented here as procedures of no arguments, and force simply calls its argument:

(define force
  (lambda (object)
    (object)))

The expression

(delay (expression))

has the same meaning as the procedure call

(make-promise (lambda () (expression)))

as follows

(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (make-promise (lambda () expression))))),

where make-promise is defined as follows:

(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result))))))

References


ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

accessor 15
antimark 51
&assertion 28
assertion-violation? 28
assoc 12
assp 12
assq 13
assv 13

base record type 15
big-endian 5
binary port 31, 32
binary-port? 34
bit fields 43
bitwise-and 49
bitwise-arithmetic-shift 50
bitwise-arithmetic-shift-left 50
bitwise-arithmetic-shift-right 50
bitwise-bit-count 49
bitwise-bit-field 49
bitwise-bit-set? 49
bitwise-copy-bit 49
bitwise-copy-bit-field 50
bitwise-first-bit-set 49
bitwise-if 49
bitwise-ior 49
bitwise-length 49
bitwise-not 49
bitwise-reverse-bit-field 50
bitwise-rotate-bit-field 50
bitwise-xor 49
bound-identifier=? 54
buffer-mode 32
buffer-mode? 32
byte 3
bytevector 3
bytevector->sint-list 7
bytevector->string 33
bytevector->u8-list 7
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-double-native-set! 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 9
bytevector-s16-native-set! 9
bytevector-s16-ref 9
bytevector-s16-set! 9
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 6
bytevector-s8-set! 6
bytevector-sint-ref 6
bytevector-sint-set! 6
bytevector-u16-native-ref 7
bytevector-u16-native-set! 7
bytevector-u16-ref 7
bytevector-u16-set! 7
bytevector-u32-native-ref 8
bytevector-u32-native-set! 8
bytevector-u32-ref 8
bytevector-u32-set! 8
bytevector-u64-native-ref 8
bytevector-u64-native-set! 8
bytevector-u64-ref 8
bytevector-u64-set! 8
bytevector-u8-ref 6
bytevector-u8-set! 6
bytevector-uint-ref 6
bytevector-uint-set! 6
bytevector=? 6
bytevector? 6

call by need 64
call-with-bytevector-output-port 39
call-with-input-file 42
call-with-output-file 42
call-with-port 35
call-with-string-output-port 39
case-lambda 15
char-alphabetic? 3
char-ci<=? 3
char-ci<= 3
char-ci?= 3
char-ci> 3
char-ci>=? 3
char-downcase 3
char-foldcase 3
char-general-category 4
char-lower-case? 3
char-numeric? 3
char-title-case? 3
char-titlecase 3
char-upcase 3
char-upper-case? 3
char-whitespace? 3
close-input-port 42
close-output-port 42
close-port 35
codec 32
command-line 43
compound condition 26
closure 26
&condition 26
closure 26
closure-accessor 26
closure-irritants 28
closure-message 28
closure-predicate 26
closure-who 29
closure? 26
cons* 13
constructor descriptor 21
continuable exception 25
current exceptional exception 24
current-error-port 40, 42
current-input-port 36, 42
current-output-port 40, 42
datum->syntax 56
define-condition-type 27
define-enumeration 62
define-record-type 16
delay 64
delete-file 43
display 43
do 14
else 25
do-end-of-file object 34
do-end-of-line 32
do-end-of-line style 32
do-endianness 5
do-endianness 5
do-endianness 5
do-endianness 5
do-endianness 5
do-endianness 5
do-endianness 5
do-endianness 5
do-endianness 5
enum-set->list 61
enum-set-complement 62
enum-set-constructor 61
enum-set-difference 61
enum-set-indexer 61
enum-set-intersection 61
enum-set-member? 61
enum-set-projection 62
enum-set-subset? 61
enum-set-union 61
enum-set-universe 61

enum-set=? 61
enum-set 61
enumeration 60
enumeration sets 61
enumeration type 61
environment 63
environment 63
eof-object 34
eof-object? 34
eof-object? 34
eol-style 32
equal-hash 60
equivalence function 59
&error 28
error-handling-mode 33
error? 28
eval 63
exact->inexact 64
exceptional situation 26
exceptions 24
exists 10
exit 43

field 15
fields 16
file options 31
file-exists? 38
file-options 31
filter 11
find 10
fixnum->flonum 19
f1* 17
f1+ 17
f1- 17
f1/ 17
f1<=? 16
f1<? 16
f1=>? 16
f1>? 16
flabs 17
flacos 18
flasin 18
flatan 18
flceiling 18
flcos 18
fldenominator 17
fldiv 17
fldiv-and-mod 17
fldiv0 17
fldiv0-and-mod0 17
fleven? 17
flexp 18
flexpt 18
ffinite? 17
ffloor 18
flinfinite? 17
Index 69

flinteger? 47
filog 48
flmax 47
flmin 47
flmod 47
flmod0 47
flnan? 47
flnegative? 47
flnumerator 47
flodd? 47
flnum? 46
flpositive? 47
flround 48
flsin 48
flsqrt 48
fltan 48
fltruncate 48
flush-output-port 39
flzero? 47
fold-left 11
fold-right 11
for-all 10
force 64
free-identifier=? 55
fx* 44
fx*/carry 45
fx+ 44
fx+/carry 44
fx- 44
fx-/carry 44
fx<=? 44
fx>? 44
fx? 44
fxand 45
fxarithmetic-shift 46
fxarithmetic-shift-left 46
fxarithmetic-shift-right 46
fxbit-count 45
fxbit-field 45
fxbit-set? 45
fxcopy-bit 45
fxcopy-bit-field 45
fxdiv 44
fxdiv-and-mod 44
fxdiv0 43
fxdiv0-and-mod0 44
fxeven? 44
fxfirst-bit-set 45
fxif 45
fxior 45
fxlength 45
fxmax 44
fxmin 44
fxmod 44
fxmod0 44
fxnegative? 44
fxnot 45
fxodd? 44
fxpositive? 44
fxreverse-bit-field 40
fxrotate-bit-field 40
fxxor 45
fxzero? 44
generate-temporaries 57
get-bytevector-all 37
get-bytevector-n 37
get-bytevector-n! 37
get-bytevector-some 37
get-char 38
get-datum 38
get-line 38
get-string-all 38
get-string-n 38
get-string-n! 38
guard 25
hash function 59
hashtable 59
hashtable-clear! 60
hashtable-contains? 60
hashtable-copy 60
hashtable-delete! 59
hashtable-entries 60
hashtable-equivalence-function 60
hashtable-hash-function 60
hashtable-keys 60
hashtable-mutable? 60
hashtable-ref 59
hashtable-set! 59
hashtable-size 60
hashtable-update! 60
hashtable? 60
&i/o 30
&i/o-decoding 33
&i/o-decoding-error? 33
&i/o-encoding 33
&i/o-encoding-error-char 33
&i/o-encoding-error? 33
&i/o-error-filename 30
&i/o-error-port 31
&i/o-error-position 30
&i/o-error? 30
&i/o-file-already-exists 30
&i/o-file-already-exists-error? 30
&i/o-file-does-not-exist 31
&i/o-file-does-not-exist-error? 31
&i/o-file-is-read-only 30
i/o-file-is-read-only-error? 30
&i/o-file-protection 30
i/o-file-protection-error? 30
&i/o-filename 30
i/o-filename-error? 30
&i/o-invalid-position 30
i/o-invalid-position-error? 30
&i/o-port 31
i/o-port-error? 31
&i/o-read 30
i/o-read-error? 30
&i/o-write 30
i/o-write-error? 30
identifier macro 54
identifier? 54
immutable 16
immutable record type 16
&implementation-restriction 29
implementation-restriction-violation? 29
implicit identifier 56
inexact->exact 43
input port 41
input-port? 35
&irritants 28
irritants-condition? 28
latin-1-codec 32
lazy evaluation 64
&lexical 29
lexical-violation? 29
list-sort 13
little-endian 5
lookahead-char 38
lookahead-u8 37
macro transformer 52
make-assertion-violation 28
make-bytevector 9
make-custom-binary-input-port 36
make-custom-binary-input/output-port 41
make-custom-binary-output-port 40
make-custom-textual-input-port 36
make-custom-textual-input/output-port 41
make-custom-textual-output-port 40
make-enumeration 61
make-eq-hashtable 59
make-eqv-hashtable 59
make-error 28
make-hashtable 59
make-i/o-decoding-error 33
make-i/o-encoding-error 33
make-i/o-error 30
make-i/o-file-already-exists-error 30
make-i/o-file-does-not-exist-error 31
make-i/o-file-is-read-only-error 30
make-i/o-file-protection-error 30
make-i/o-filename-error 30
make-i/o-invalid-position-error 30
make-i/o-port-error 31
make-i/o-read-error 30
make-i/o-write-error 30
make-implementation-restriction-violation 29
make-irritants-condition 28
make-lexical-violation 29
make-message-condition 28
make-no-infinities-violation 48
make-no-nans-violation 48
make-non-continuable-violation 29
make-record-constructor-descriptor 21
make-record-type-descriptor 20
make-serious-condition 28
make-syntax-violation 29
make-transcoder 33
make-undefined-violation 29
make-variable-transformer 52
make-violation 28
make-warning 28
make-who-condition 29
mark 51
member 12
memp 12
memq 12
memv 12
&message 28
message-condition? 28
modulo 44
mutable 10
mutable record type 10
mutator 15
native-endianness 6
native-eol-style 33
native-transcoder 33
newline 43
&no-infinities 48
no-infinities-violation? 48
&no-nans 48
no-nans-violation? 48
&non-continuable 29
non-continuable-violation? 29
nongenerative 16
nongenerative 16
null-environment 65
number 43
octet 5
opaque 16
open-bytevector-input-port 45
open-bytevector-output-port 39
open-file-input-port 35
open-file-input/output-port 41
string-ci-hash 60
string-ci<? 4
string-ci<=? 4
string-ci=? 4
string-ci>=? 4
string-ci>? 4
string-downcase 4
string-fill! 64
string-foldcase 4
string-hash 60
string-normalize-nfc 4
string-normalize-nfd 4
string-normalize-nfkc 4
string-normalize-nfkd 4
string-set! 63
string-titlecase 4
string-upcase 4
substitution 51
symbol-hash 60
&syntax 29
syntax 23
syntax object 51
syntax->datum 50
syntax-case 52
syntax-violation 58
syntax-violation-form 29
syntax-violation-subform 29
syntax-violation? 29
textual port 32
textual ports 31
textual-port? 34
transcoded-port 34
transcoder 32
transcoder-codec 33
transcoder-eol-style 33
transcoder-error-handling-mode 34
transformation procedure 52
transformer 52
u8-list->bytevector 6
uint-list->bytevector 7
&undefined 29
undefined-violation? 29
universe 61
unless 13
unsyntax 57
unsyntax-splicing 57
utf-16-codec 32
utf-8-codec 52
utf16->string 10
utf32->string 10
utf8->string 9
variable transformer 52
vector-sort 13
vector-sort! 13
&violation 28
violation? 28
&warning 28
warning? 28
when 14
&who 29
who-condition? 29
with-exception-handler 24
with-input-from-file 42
with-output-to-file 42
with-syntax 57
wrap 51
wrapped syntax object 51
write 13
write-char 12