Revised$^{5.91}$ Report on the Algorithmic Language Scheme

Michael Sperber
William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten
Richard Kelsey, Jonathan Rees
(*Editors*)

H. Abelson, N. I. Adams iv, D. H. Bartley, G. Brooks, R. B. Findler, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, J. Matthews, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele Jr., G. J. Sussman, M. Wand

5 September 2006

SUMMARY

The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.

The introduction offers a brief history of the language and of the report. It also gives a short introduction to the basic concepts of the language.

Chapter 2 explains Scheme's number types. Chapter 3 defines the read syntax of Scheme programs. Chapter 4 presents the fundamental semantic ideas of the language. Chapter 5 defines notational conventions used in the rest of the report. Chapters 6 and 7 describe libraries and scripts, the basic organizational units of Scheme programs. Chapter 8 explains the expansion process for Scheme code.

Chapter 9 explains the Scheme base library which contains the fundamental forms useful to programmers.

The next set of chapters describe libraries that provide specific functionality: Unicode semantics for characters and strings, binary data, list utility procedures, a record system, exceptions and conditions, I/O, specialized libraries for dealing with numbers and arithmetic, the `syntax-case` facility for writing arbitrary macros, hash tables, enumerations, and various miscellaneous libraries.

Chapter 21 describes the composite library containing most of the forms described in this report. Chapter 22 describes the `eval` facility for evaluating Scheme expressions represented as data. Chapter 23 describes the operations for mutating pairs. Chapter 24 describes a library with some procedures from the previous version of this report for backwards compatibility.

Appendix A provides a formal semantics for a core of Scheme. Appendix B contains definitions for some of the derived forms described in the report.

The report concludes with a list of references and an alphabetic index.

*** DRAFT***

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

CONTENTS

INTRODUCTION

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially goto's that pass arguments. Scheme was the first widely used programming language to embrace first class escape procedures, from which all previously known sequential control structures can be synthesized. A subsequent version of Scheme introduced the concept of exact and inexact numbers, an extension of Common Lisp's generic arithmetic. More recently, Scheme became the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended in a consistent and reliable manner.

Numerical computation was long neglected by the Lisp community. Until Common Lisp there was no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system [39] little effort was made to execute numerical code efficiently. The Scheme reports recognized the excellent work of the Common Lisp committee and accepted many of their recommendations, while simplifying and generalizing in some ways consistent with the purposes of Scheme.

Background

The first description of Scheme was written by Gerald Jay Sussman and Guy Lewis Steele Jr. in 1975 [47]. A revised report by Steele and Sussman [44] appeared in 1978 and described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [45]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [40, 36, 19]. An introductory computer science textbook using Scheme was published in 1984 [1]. A number of textbooks describing und using Scheme have been published since [15].

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Their report [7], edited by Will Clinger, was published at MIT and Indiana University in the summer of 1985. Further revision took place in the spring of 1986 [42] (edited by Jonathan Rees and Will Clinger), and in the spring of 1988 [9] (also edited by Will Clinger and Jonathan Rees). Another revision published in 1998, edited by Richard Kelsey, Will Clinger and Jonathan Rees, reflected further revisions agreed upon in a meeting at Xerox PARC in June 1992 [28].

Attendees of the Scheme Workshop in Pittsburgh in October 2002 formed a Strategy Committee to discuss a process for producing new revisions of the report. The strategy committee drafted a charter for Scheme standardization. This charter, together with a process for selecting editors' committees for producing new revisions for the report, was confirmed by the attendees of the Scheme Workshop in Boston in November 2003. Subsequently, a Steering Committee according to the charter was selected, consisting of Alan Bawden, Guy L. Steele Jr., and Mitch Wand. An editors' committee charged with producing this report was also formed at the end of 2003, consisting of Will Clinger, R. Kent Dybvig, Marc Feeley, Matthew Flatt, Richard Kelsey, Manuel Serrano, and Mike Sperber, with Marc Feeley acting as Editor-in-Chief. Richard Kelsey resigned from the committee in April 2005, and was replaced by Anton van Straaten. Marc Feeley and Manuel Serrano resigned from the committee in January 2006. Subsequently, the charter was revised to reduce the size of the editors' committee to five and to replace the office of Editor-in-Chief by a Chair and a Project Editor [48]. R. Kent Dybvig served as Chair, and Mike Sperber served as Project Editor. Parts of the report were posted as Scheme Requests for Implementation (SRFIs) and discussed by the community before being revised and finalized for the report [22, 12, 13, 21, 17]. Jacob Matthews and Robby Findler wrote the operational semantics for the language core.

Acknowledgements

We would like to thank the following people for their help: Alan Bawden, Michael Blair, Per Bothner, Thomas

Bushnell, Taylor Campbell, John Cowan, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Ray Dillinger, Bruce Duba, Sebastian Egner, Tom Emerson, Marc Feeley, Andy Freeman, Richard Gabriel, Martin Gasbichler, Arthur A. Gleckler, Aziz Ghuloum, Yekta Gürsel, Ken Haase, Lars T Hansen, Robert Hieb, Paul Hudak, Aubrey Jaffer, Shiro Kawai, Morry Katz, Donovan Kolbly, Chris Lindblad, Bradley Lucier, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Jorgen Schaefer, Paul Schlie, Manuel Serrano, Mike Shaff, Olin Shivers, Jonathan Shapiro, Julie Sussman, David Van Horn, Andre van Tonder, Oscar Waddell, Perry Wagle, Alan Watson, Daniel Weise, Andrew Wilcox, Henry Wu, and Ozan Yigit. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual* [50]. We gladly acknowledge the influence of manuals for MIT Scheme [36], T [41], Scheme 84 [23], Common Lisp [46], Chez Scheme [16], PLT Scheme [20], and Algol 60 [37].

We also thank Betty Dexter for the extreme effort she put into setting this report in TEX, and Donald Knuth for designing the program that caused her troubles.

The Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, the Computer Science Department of Indiana University, the Computer and Information Sciences Department of the University of Oregon, and the NEC Research Institute supported the preparation of this report. Support for the MIT work was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Support for the Indiana University work was provided by NSF grants NCS 83-04567 and NCS 83-03325.

DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

This chapter gives an overview of Scheme's semantics. A detailed informal semantics is the subject of the following chapters. For reference purposes, appendix A provides a formal semantics for a core subset of Scheme.

Following Algol, Scheme is a statically scoped programming language. Each use of a variable is associated with a lexically apparent binding of that variable.

Scheme has latent as opposed to manifest types [53]. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are Python, Ruby, Smalltalk, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, C, C#, Java, Haskell and ML.

All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include C#, Haskell, ML, Python, Ruby, Smalltalk and other Lisp dialects.

Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar. See section 4.7.

Scheme was one of the first languages to support procedures as objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp, Haskell, ML, Smalltalk, and Ruby.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section 9.18.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. C, C#, Common Lisp, Python, Ruby, and Smalltalk are other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure. Note that call-by-value refers to a different distinction than the distinction between by-value and by-reference passing in Pascal. In Scheme, all data structures are passed by-reference.

Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Scheme, every integer is a rational number, every rational is a real, and every real is a complex number. Scheme distinguishes between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. Exact arithmetic includes arithmetic on integers, rationals and complex numbers.

The following sections give a brief overview of the most fundamental elements of the language. The purpose of this overview is to explain enough of the basic concepts of the language to facilitate understanding of the subsequent chapters of the report, which are organized as a reference manual. Consequently, this overview is not a complete introduction of the language, nor is it precise in all respects.

1.1. Basic types

Scheme programs manipulate *values*, which are also referred to as *objects*. Scheme values are organized into sets of values called *types*. This gives an overview of the fundamentally important types of the Scheme language. More types are described in later chapters.

Note: As Scheme is latently typed, the use of the term *type* in this report differs from the use of the term in the context of other languages, particularly those with manifest typing.

Boolean values A boolean value denotes a truth value, and can either be true or false. In Scheme, the value for "false" is written `#f`. The value "true" is written `#t`. In most places where a truth value is expected, however, any value different from `#f` counts as true.

Numbers Scheme supports a rich variety of numerical data types, including integers of arbitrary precision, rational numbers, complex numbers and inexact numbers of various kinds. Chapter 2 gives an overview of the structure of Scheme's numerical tower.

Characters Scheme characters mostly correspond to textual characters. More precisely, they are isomorphic to the *scalar values* of the Unicode standard.

Strings Strings are finite sequences of characters with fixed length and thus represent arbitrary Unicode texts.

Symbols A symbol is an object representing a string that cannot be modified. This string is called the symbol's *name*. Unlike strings, two symbols whose names are spelled the same way are indistinguishable. Symbols are useful for many applications; for instance, they may be used the way enumerated values are used in other languages.

Pairs and lists A pair is a data structure with two components. The most common use of pairs is to represent (singly linked) lists, where the first component (the "car") represents the first element of the list, and the second component (the "cdr") the rest of the list. Scheme also has a distinguished empty list, which is the last cdr in a chain of pairs representing a list.

Vectors Vectors, like lists, are linear data structures representing finite sequences of arbitrary objects. Whereas the elements of a list are accessed sequentially through the pair chain representing it, the elements of a vector are addressed by an integer index. Thus, vectors are more appropriate than lists for random access to elements.

Procedures As mentioned in the introduction, procedures are values in Scheme.

1.2. Expressions

The most important elements of a Scheme program are *expressions*. Expressions can be *evaluated*, producing a *value*. (Actually, any number of values—see section 4.5.) The most fundamental expressions are literal expressions:

```
#t                              ⟹ #t
23                              ⟹ 23
```

This notation means that the expression `#t` evaluates to `#t`, that is, the value for "true," and that the expression `23` evaluates to the number 23.

Compound expressions are formed by placing parentheses around their subexpressions. The first subexpression is an *operator* and identifies an operation; the remaining subexpressions are *operands*:

```
(+ 23 42)                       ⟹ 65
(+ 14 (* 23 42))                ⟹ 980
```

In the first of these examples, `+`, the operator, is the name of the built-in operation for addition, and `23` and `42` are the operands. The expression `(+ 23 42)` reads as "the sum of 23 and 42." Compound expressions can be nested—the second example reads as "the sum of 14 and the product of 23 and 42."

As these examples indicate, compound expressions in Scheme are always written using the same prefix notation. As a consequence, the parentheses are needed to indicate structure, and "superfluous" parentheses, which are permissible in mathematics and many programming languages, are not allowed in Scheme.

As in many other languages, whitespace and newlines are not significant when they separate subexpressions of an expression, and can be used to indicate structure.

1.3. Variables and binding

Scheme allows identifiers to denote values. These identifiers are called variables. (More precisely, variables denote locations. This distinction is not important, however, for a large proportion of Scheme code.)

```
(let ((x 23)
      (y 42))
  (+ x y))                      ⟹ 65
```

In this case, the operator of the expression, `let`, is a binding construct. The parenthesized structure following the `let` lists variables alongside expressions: the variable `x` alongside `23`, and the variable `y` alongside `42`. The `let` expression binds `x` to 23, and `y` to 42. These bindings are available in the *body* of the `let` expression, `(+ x y)`, and only there.

1.4. Definitions

The variables bound by a `let` expression are *local*, because their bindings are visible only in the `let`'s body. Scheme also allows creating top-level bindings for identifiers as follows:

```
(define x 23)
(define y 42)
(+ x y)                         ⟹ 65
```

(These are actually "top-level" in the body of a library or script; see section 1.10 below.)

The first two parenthesized structures are *definitions*; they create top-level bindings, binding `x` to 23 and `y` to 42. Definitions are not expressions, and cannot appear in all places where an expression can occur. Moreover, a definition has no value.

Bindings follow the lexical structure of the program: When several bindings with the same name exist, a variable refers to the binding that is closest to it, starting with its occurrence in the program and going from inside to outside, going all the way to a top-level binding only if no local binding can be found along the way:

```
(define x 23)
(define y 42)
(let ((y 43))
  (+ x y))                    ⟹ 66

(let ((y 43))
  (let ((y 44))
    (+ x y)))                 ⟹ 67
```

1.5. Procedures

Definitions can also be used to define procedures:

```
(define (f x)
  (+ x 42))

(f 23)                        ⟹ 65
```

A procedure is, slightly simplified, an abstraction over an expression. In the example, the first definition defines a procedure called `f`. (Note the parentheses around `f x`, which indicate that this is a procedure definition.) The expression `(f 23)` is a procedure call, meaning, roughly, "evaluate `(+ x 42)` (the body of the procedure) with `x` bound to 23."

As procedures are regular values, they can be passed to other procedures:

```
(define (f x)
  (+ x 42))

(define (g p x)
  (p x))

(g f 23)                      ⟹ 65
```

In this example, the body of `g` is evaluated with `p` bound to `f` and `x` bound to 23, which is equivalent to `(f 23)`, which evaluates to 42.

In fact, many predefined operations of Scheme are bindings for procedures. `+`, for example, which receives special syntactic treatment in many other languages, is just a regular identifier in Scheme, bound to a procedure that adds numbers. The same holds for `*` and many others:

```
(define (h op x y)
  (op x y))

(h + 23 42)                   ⟹ 65
(h * 23 42)                   ⟹ 966
```

Procedure definitions are not the only way to create procedures. A `lambda` expression creates a new procedure as a value, with no need to specify a name:

```
((lambda (x) (+ x 42)) 23)  ⟹ 65
```

The entire expression in this example is a procedure call; its operator is `(lambda (x) (+ x 42))`, which evaluates to a procedure that takes a single number and add it to 42.

1.6. Procedure calls and syntactic keywords

Whereas `(+ 23 42)`, `(f 23)`, and `((lambda (x) (+ x 42)) 23)` are all examples of procedure calls, `lambda` and `let` expressions are not. This is because `let`, even though it is an identifier, is not a variable, but is instead a *syntactic keyword*. An expression that has a syntactic keyword as its operator obeys special rules determined by the keyword. The `define` identifier in a definition is also a syntactic keyword. Hence, definitions are also not procedure calls.

In the case of `lambda`, these rules specify that the first subform is a list of parameters, and the second subform is the body of the procedure. In `let` expressions, the first subform is a list of binding specifications, and the second is a body of expressions.

Procedure calls can be distinguished from these "special forms" by looking for a syntactic keyword in the first position of an expression: if it is not a syntactic keyword, the expression is a procedure call. The set of syntactic keywords of Scheme is fairly small, which usually makes this task fairly simple. It is possible, however, to create new bindings for syntactic keywords; see below.

1.7. Assignment

Scheme variables bound by definitions or `let` or `lambda` forms are not actually bound directly to the values specified in the respective bindings, but to locations containing these values. The contents of these locations can subsequently be modified destructively via *assignment*:

```
(let ((x 23))
  (set! x 42)
  x)                          ⟹ 42
```

In this case, the body of the `let` expression consists of two expressions which are evaluated sequentially, with the value of the final expression becoming the value of the entire `let` expression. The expression `(set! x 42)` is an assignment, saying "replace the value in the location denoted by `x` with 42." Thus, the previous value of 23 is replaced by 42.

1.8. Derived forms and macros

Many of the special forms specified in this report can be translated into more basic special forms. For example, `let` expressions can be translated into procedure calls and `lambda` expressions. The following two expressions are equivalent:

```
(let ((x 23)
      (y 42))
  (+ x y))                    ⟹ 65

((lambda (x y) (+ x y)) 23 42)
          ⟹ 65
```

Special forms like `let` expressions are called *derived forms* because their semantics can be derived from that of other kinds of forms by a syntactic transformation. Procedure definitions are also derived forms. The following two definitions are equivalent:

```
(define (f x)
  (+ x 42))

(define f
  (lambda (x)
    (+ x 42)))
```

In Scheme, it is possible for a program to create its own derived forms by binding syntactic keywords to macros:

```
(define-syntax def
  (syntax-rules ()
    ((def f (p ...) body)
     (define (f p ...)
       body))))

(def f (x)
  (+ x 32))
```

The `define-syntax` construct specifies that a parenthesized structure matching the pattern `(def f (p ...) body)`, where `f`, `p`, and `body` are pattern variables, is translated to `(define (f p ...) body)`. Thus, the `def` form appearing in the example gets translated to:

```
(define (f x)
  (+ x 42))
```

The ability to create new syntactic keywords makes Scheme extremely flexible and expressive, enabling the formulation of many features built into other languages as derived forms.

1.9. Syntactic datums and datum values

A subset of the Scheme values called *datum values* have a special status in the language. These include booleans, numbers, characters, and strings as well as lists and vectors whose elements are datums. Each datum value may be represented in textual form as a *syntactic datum*, which can be written out and read back in without loss of information. Several syntactic datums can represent the same datum value, but the datum value corresponding to a syntactic datum is uniquely determined. Moreover, each datum value can be trivially translated to a literal expression in a program by prepending a `'` to a corresponding syntactic datum:

```
'23                           ⟹ 23
'#t                           ⟹ #t
'foo                          ⟹ foo
'(1 2 3)                      ⟹ (1 2 3)
'#(1 2 3)                     ⟹ #(1 2 3)
```

The `'` is, of course, not needed for number or boolean literals. The identifier `foo` is a syntactic datum that can represent a symbol with name "foo," and `'foo` is a literal expression with that symbol as its value. `(1 2 3)` is a syntactic datum that can represent a list with elements 1, 2, and 3, and `'(1 2 3)` is a literal expression with this list as its value. Likewise, `#(1 2 3)` is a syntactic datum that can represent a vector with elements 1, 2 and 3, and `'#(1 2 3)` is the corresponding literal.

The syntactic datums form a superset of the Scheme forms. Thus, datums can be used to represent Scheme programs as data objects. In particular, symbols can be used to represent identifiers.

```
'(+ 23 42)                    ⟹ (+ 23 42)
'(define (f x) (+ x 42))
          ⟹ (define (f x) (+ x 42))
```

This facilitates writing programs that operate on Scheme source code, in particular intpreters and program transformers.

1.10. Libraries

Scheme code is organized in components called *libraries*. Each library contains declarations, definitions and expressions. It can import definitions from other libraries, and export definitions to other libraries:

```
(library (hello)
  (export)
  (import (r6rs base)
          (r6rs i/o simple))
  (display "Hello World")
  (newline))
```

1.11. Scripts

A Scheme program is invoked via a *script*. Like a library, a script contains declarations, definitions and expressions, but specifies an entry point for execution. Thus, a script defines, via the transitive closure of the libraries it imports, a Scheme program.

```
#! /usr/bin/env scheme-script
#!r6rs
(import (r6rs base)
        (r6rs i/o ports))
(put-bytes (standard-output-port)
           (call-with-port
               (open-file-input-port
                 (cadr (command-line-arguments)))
             get-bytes-all))
0
```

2. Numbers

This chapter describes Scheme's representations for numbers. It is important to distinguish between the mathematical numbers, the Scheme numbers that attempt to model them, the machine representations used to implement the Scheme numbers, and notations used to write numbers. This report uses the types *number*, *complex*, *real*, *rational*, and *integer* to refer to both mathematical numbers and Scheme numbers. The *fixnum* and *flonum* types refer to certain subtypes of the Scheme numbers, as explained below.

2.1. Numerical types

Mathematically, numbers may be arranged into a tower of subtypes in which each level is a subset of the level above it:

number
complex
real
rational
integer

For example, 5 is an integer. Therefore 5 is also a rational, a real, and a complex. The same is true of the Scheme numbers that model 5. For Scheme numbers, these types are defined by the predicates `number?`, `complex?`, `real?`, `rational?`, and `integer?`.

There is no simple relationship between a number's type and its representation inside a computer. Although most implementations of Scheme offer at least three different representations of 5, these different representations denote the same integer.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use many different representations for numbers, this should not be apparent to a casual programmer writing simple programs.

It is necessary, however, to distinguish between numbers that are represented exactly and those that may not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

A *fixnum* is an exact integer whose value lies within a certain implementation-dependent subrange of the exact integers (section 16.3). Likewise, every implementation is required to designate a subset of its inexact reals as *flonum*s, and to convert certain external representations into flonums. Note that this does not imply that an implementation is required to use floating point representations.

2.2. Exactness

Scheme numbers are either *exact* or *inexact*. A number is exact if it is written as an exact constant or was derived from exact numbers using only exact operations. A number is inexact if it is written as an inexact constant or was derived from inexact numbers. Thus inexactness is contagious.

Exact arithmetic is reliable in the following sense: If exact numbers are passed to any of the arithmetic procedures described in section 9.10, and an exact number is returned, then the result is mathematically correct. This is generally not true of computations involving inexact numbers because approximate methods such as floating point arithmetic may be used, but it is the duty of each implementation to make the result as close as practical to the mathematically ideal result.

2.3. Implementation restrictions

Implementations of Scheme are required to implement the whole tower of subtypes given in section 2.1.

Implementations are required to support exact integers and exact rationals of practically unlimited size and precision, and to implement certain procedures (listed in 9.10.1) so they always return exact results when given exact arguments.

Implementations may support only a limited range of inexact numbers of any type, subject to the requirements of this section. For example, an implementation may limit the range of inexact reals (and therefore the range of inexact integers and rationals) to the dynamic range of the flonum format. Furthermore the gaps between the representable inexact integers and rationals are likely to be very large in such an implementation as the limits of this range are approached.

An implementation may use floating point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that the IEEE floating point standards be followed by implementations that use floating point representations, and that implementations using other representations should match or exceed the precision achievable using these floating point standards [26].

In particular, implementations that use floating point representations must follow these rules: A floating point result must be represented with at least as much precision as is used to express any of the inexact arguments to that operation. It is desirable (but not required) for potentially inexact operations such as `sqrt`, when applied to exact arguments, to produce exact answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If, however, an exact number is operated upon so as to produce an inexact result (as by `sqrt`), and if the result is represented in floating point, then the most precise floating point format available must be used; but if the result is represented in some other way then the representation must have at least as much precision as the most precise floating point format available.

It is the programmer's responsibility to avoid using inexact numbers with magnitude or significand too large to be represented in the implementation.

2.4. Infinities and NaNs

Positive infinity is regarded as a real (but not rational) number, whose value is indeterminate but greater than all rational numbers. Negative infinity is regarded as a real (but not rational) number, whose value is indeterminate but less than all rational numbers.

A NaN is regarded as a real (but not rational) number whose value is so indeterminate that it might represent any real number, including positive or negative infinity, and might even be greater than positive infinity or less than negative infinity.

3. Lexical syntax and read syntax

The syntax of Scheme programs is organized in three levels:

1. the *lexical syntax* that describes how a program text is split into a sequence of lexemes,
2. the *read syntax*, formulated in terms of the lexical syntax, that structures the lexeme sequence as a sequence of *syntactic datums*, where a syntactic datum is a recursively structured entity,
3. the *program syntax* formulated in terms of the read syntax, imposing further structure and assigning meaning to syntactic datums.

Syntactic datums (also called *external representations*) double as a notation for data, and Scheme's `(r6rs i/o ports)` library (section 15.3) provides the `get-datum` and `put-datum` procedures for reading and writing syntactic datums, converting between their textual representation and the corresponding values. A syntactic datum can be used in a program to obtain the corresponding value using `quote` (see section 9.5.1).

Moreover, valid Scheme expressions form a subset of the syntactic datums. Consequently, Scheme's syntax has the property that any sequence of characters that is an expression is also a syntactic datum representing some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program. It is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa). A syntactic datum occurring in program text is often called a *form*.

Note that several syntactic datums may represent the same object, a so-called *datum value*. For example, both"`#e28.000`" and "`#x1c`" are syntactic datums representing the exact integer 28; The syntactic datums "`(8 13)`", "`( 08 13 )`", "`(8 . (13 . ()))`" (and more) all represent a list containing the integers 8 and 13. Syntactic datums that denote equal objects are always equivalent as forms of a program.

Because of the close correspondence between syntactic datums and datum values, this report sometimes uses the term *datum* to denote either a syntactic datum or a datum value when the exact meaning is apparent from the context.

An implementation is not permitted to extend the lexical or read syntax in any way, with one exception: it need not treat the syntax `#!`⟨identifier⟩, for any ⟨identifier⟩ (see section 3.2.3) that is not `r6rs`, as a syntax violation, and it may use specific `#!`-prefixed identifiers as flags indicating that subsequent input contains extensions to the standard lexical syntax. (The comment syntax `#!r6rs` may be used to signify that the input which follows is written purely in the language described by this report; see section 3.2.2.)

This chapter overviews and provides formal accounts of the lexical syntax and the read syntax.

3.1. Notation

The formal syntax for Scheme is written in an extended BNF. Non-terminals are written using angle brackets; case is insignificant for non-terminal names.

All spaces in the grammar are for legibility. ⟨Empty⟩ stands for the empty string.

The following extensions to BNF are used to make the description more concise: ⟨thing⟩* means zero or more occurrences of ⟨thing⟩; and ⟨thing⟩$^+$ means at least one ⟨thing⟩.

Some non-terminal names refer to the Unicode scalar values of the same name: ⟨character tabulation⟩ (U+0009), ⟨linefeed⟩ (U+000A), ⟨line tabulation⟩ (U+000B), ⟨form feed⟩ (U+000C), ⟨carriage return⟩ (U+000D), and ⟨space⟩ (U+0020).

3.2. Lexical syntax

The lexical syntax describes how a character sequence is split into a sequence of lexemes, omitting non-significant portions such as comments and whitespace. The character sequence is assumed to be text according to the Unicode standard [51]. Some of the lexemes, such as numbers, identifiers, strings etc. of the lexical syntax are syntactic datums in the read syntax, and thus represent data. Besides the formal account of the syntax, this section also describes what datum values are denoted by these syntactic datums.

Note that the lexical syntax, in the description of comments, contains a forward reference to ⟨datum⟩, which is described as part of the read syntax. However, being comments, these ⟨datum⟩s do not play a significant role in the syntax.

Case is significant except in boolean datums, number datums, and hexadecimal numbers denoting scalar values. For example, `#x1A` and `#X1a` are equivalent. The identifier `Foo` is, however, distinct from the identifier `FOO`.

3.2.1. Formal account

⟨Interlexeme space⟩ may occur on either side of any lexeme, but not within a lexeme.

Lexemes that require implicit termination (identifiers, numbers, characters, booleans, and dot) are terminated by any ⟨delimiter⟩ or by the end of the input, but not necessarily by anything else.

The following two characters are reserved for future extensions to the language: `{ }`

⟨lexeme⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
    | `(` | `)` | `[` | `]` | `#(` | `'` | `` ` `` | `,` | `,@` | `.`
⟨delimiter⟩ ⟶ ⟨whitespace⟩ | `(` | `)` | `[` | `]` | `"` | `;`
⟨whitespace⟩ ⟶ ⟨character tabulation⟩ | ⟨linefeed⟩
    | ⟨line tabulation⟩ | ⟨form feed⟩ ⟨carriage return⟩
    | ⟨any character whose category is Zs, Zl, or Zp⟩
⟨intra-line whitespace⟩ ⟶ ⟨any ⟨whitespace⟩
      that is not ⟨linefeed⟩⟩
⟨comment⟩ ⟶ `;` ⟨all subsequent characters up to a
      linefeed⟩
    | ⟨nested comment⟩
    | `#;` ⟨datum⟩
    | `#!r6rs`
⟨nested comment⟩ ⟶ `#|` ⟨comment text⟩
      ⟨comment cont⟩* `|#`
⟨comment text⟩ ⟶ ⟨character sequence not containing
      `#|` or `|#`⟩
⟨comment cont⟩ ⟶ ⟨nested comment⟩ ⟨comment text⟩
⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩
⟨interlexeme space⟩ ⟶ ⟨atmosphere⟩*

⟨identifier⟩ ⟶ ⟨initial⟩ ⟨subsequent⟩*
    | ⟨peculiar identifier⟩
⟨initial⟩ ⟶ ⟨constituent⟩ | ⟨special initial⟩
    | ⟨symbol escape⟩
⟨letter⟩ ⟶ `a` | `b` | `c` | `...` | `z`
    | `A` | `B` | `C` | `...` | `Z`
⟨constituent⟩ ⟶ ⟨letter⟩
    | ⟨any character whose scalar value is greater than
      127, and whose category is Lu, Lt, Lm, Lo, Mn, Mc,
      Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co⟩

⟨special initial⟩ ⟶ `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=`
    | `>` | `?` | `^` | `_` | `~`
⟨subsequent⟩ ⟶ ⟨initial⟩ | ⟨digit⟩
    | ⟨special subsequent⟩
    | ⟨inline hex escape⟩
⟨digit⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨hex digit⟩ ⟶ ⟨digit⟩
    | `a` | `A` | `b` | `B` | `c` | `C` | `d` | `D` | `e` | `E` | `f` | `F`
⟨special subsequent⟩ ⟶ `+` | `-` | `.` | `@`
⟨inline hex escape⟩ ⟶ `\x`⟨hex scalar value⟩`;`
⟨hex scalar value⟩ ⟶ ⟨hex digit⟩$^+$
    with at most 8 digits
⟨peculiar identifier⟩ ⟶ `+` | `-` | `...` | `->` ⟨subsequent⟩*
⟨boolean⟩ ⟶ `#t` | `#T` | `#f` | `#F`
⟨character⟩ ⟶ `#\`⟨any character⟩
    | `#\`⟨character name⟩
    | `#\x`⟨hex scalar value⟩
⟨character name⟩ ⟶ `nul` | `alarm` | `backspace` | `tab`
    | `linefeed` | `vtab` | `page` | `return` | `esc`
    | `space` | `delete`

⟨string⟩ ⟶ `"` ⟨string element⟩* `"`
⟨string element⟩ ⟶ ⟨any character other than `"` or `\`⟩
    | `\a` | `\b` | `\t` | `\n` | `\v` | `\f` | `\r`
    | `\"` | `\\`
    | `\`⟨linefeed⟩ | `\`⟨space⟩
    | ⟨inline hex escape⟩

⟨number⟩ ⟶ ⟨num 2⟩ | ⟨num 8⟩
    | ⟨num 10⟩ | ⟨num 16⟩

The following rules for ⟨num R⟩, ⟨complex R⟩, ⟨real R⟩, ⟨ureal R⟩, ⟨uinteger R⟩, and ⟨prefix R⟩ should be replicated for $R = 2, 8, 10$, and 16. There are no rules for ⟨decimal 2⟩, ⟨decimal 8⟩, and ⟨decimal 16⟩, which means that numbers containing decimal points or exponents must be in decimal radix.

⟨num R⟩ ⟶ ⟨prefix R⟩ ⟨complex R⟩
⟨complex R⟩ ⟶ ⟨real R⟩ | ⟨real R⟩ `@` ⟨real R⟩
| ⟨real R⟩ `+` ⟨ureal R⟩ `i` | ⟨real R⟩ `-` ⟨ureal R⟩ `i`
| ⟨real R⟩ `+ i` | ⟨real R⟩ `- i`
| `+` ⟨ureal R⟩ `i` | `-` ⟨ureal R⟩ `i` | `+ i` | `- i`
⟨real R⟩ ⟶ ⟨sign⟩ ⟨ureal R⟩
⟨ureal R⟩ ⟶ ⟨uinteger R⟩
| ⟨uinteger R⟩ `/` ⟨uinteger R⟩
| ⟨decimal R⟩ ⟨mantissa width⟩
| `inf.0` | `nan.0`
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
| `.` ⟨digit 10⟩$^+$ `#`* ⟨suffix⟩
| ⟨digit 10⟩$^+$ `.` ⟨digit 10⟩* `#`* ⟨suffix⟩
| ⟨digit 10⟩$^+$ `#`$^+$ `.` `#`* ⟨suffix⟩
⟨uinteger R⟩ ⟶ ⟨digit R⟩$^+$ `#`*
⟨prefix R⟩ ⟶ ⟨radix R⟩ ⟨exactness⟩
| ⟨exactness⟩ ⟨radix R⟩

⟨suffix⟩ ⟶ ⟨empty⟩
| ⟨exponent marker⟩ ⟨sign⟩ ⟨digit 10⟩$^+$
⟨exponent marker⟩ ⟶ `e` | `E` | `s` | `S` | `f` | `F`
| `d` | `D` | `l` | `L`
⟨mantissa width⟩ ⟶ ⟨empty⟩
| `|` ⟨digit 10⟩$^+$
⟨sign⟩ ⟶ ⟨empty⟩ | `+` | `-`
⟨exactness⟩ ⟶ ⟨empty⟩
| `#i` | `#I` | `#e` | `#E`
⟨radix 2⟩ ⟶ `#b` | `#B`
⟨radix 8⟩ ⟶ `#o` | `#O`
⟨radix 10⟩ ⟶ ⟨empty⟩ | `#d` | `#D`
⟨radix 16⟩ ⟶ `#x` | `#X`
⟨digit 2⟩ ⟶ `0` | `1`
⟨digit 8⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨hex digit⟩

3.2.2. Whitespace and comments

Whitespace characters are spaces, linefeeds, carriage returns, character tabulations, form feeds, line tabulations, and any other character whose category is Zs, Zl, or Zp. Whitespace is used for improved readability and as necessary to separate lexemes from each other. Whitespace may occur between any two lexemes, but not within a lexeme. Whitespace may also occur inside a string, where it is significant.

The lexical syntax includes several comment forms. In all cases, comments are invisible to Scheme, except that they act as delimiters, so a comment cannot appear in the middle of an identifier or number.

A semicolon (`;`) indicates the start of a line comment. The comment continues to the end of the line on which the semicolon appears (i.e., it is terminated by a linefeed character).

Another way to indicate a comment is to prefix a ⟨datum⟩ (cf. Section 3.3.1) with `#;`, possibly with whitespace before the ⟨datum⟩. The comment consists of the comment prefix `#;` and the ⟨datum⟩ together. (This notation is useful for "commenting out" sections of code.)

Block comments may be indicated with properly nested `#|` and `|#` pairs.

```
#|
   The FACT procedure computes the factorial
   of a non-negative integer.
|#
(define fact
  (lambda (n)
    ;; base case
    (if (= n 0)
        #;(= n 1)
        1       ; identity of *
        (* n (fact (- n 1))))))
```

The lexeme `#!r6rs` is also a comment. When it occurs in program text, it signifies that program text to be written purely in the language described by this report (see section 6.1).

3.2.3. Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. In particular, a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, `+`, `-`, and `...` are identifiers. Here are some examples of identifiers:

```
lambda                    q
list->vector              soup
+                         V17a
<=?                       a34kTMNs
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

Moreover, all characters whose scalar values are greater than 127 and whose Unicode category is Lu, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co can be used within identifiers. Moreover, any character can

appear as the constituent of an identifier when denoted via a hexadecimal escape sequence. For example, the identifier `H\x65;llo` is the same as the identifier `Hello`, and the identifier `\x3BB;` is the same as the identifier λ.

Any identifier may be used as a variable or as a syntactic keyword (see sections 4.2 and 6.3.2) in a Scheme program.

Moreover, when viewed as a datum value, an identifier denotes a *symbol* (see section 9.13).

3.2.4. Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. The character after a boolean literal must be a delimiter character, such as a space or parenthesis.

3.2.5. Characters

Characters are written using the notation `#\`⟨character⟩ or `#\`⟨character name⟩ or `#\x`⟨digit 16⟩$^+$, where the last specifies the scalar value of a character with a hexadecimal number of no more than eight digits.

For example:

```
#\a             ⟹ lower case letter a
#\A             ⟹ upper case letter A
#\(             ⟹ left parenthesis
#\              ⟹ space character
#\nul           ⟹ U+0000
#\alarm         ⟹ U+0007
#\backspace     ⟹ U+0008
#\tab           ⟹ U+0009
#\linefeed      ⟹ U+000A
#\vtab          ⟹ U+000B
#\page          ⟹ U+000C
#\return        ⟹ U+000D
#\esc           ⟹ U+001B
#\space         ⟹ U+0020
                ; preferred way to write a space
#\delete        ⟹ U+007F

#\xFF           ⟹ U+00FF
#\x03BB         ⟹ U+03BB
#\x00006587     ⟹ U+6587
#\λ             ⟹ U+03BB

#\x0001z        ⟹ &lexical exception
#\λx            ⟹ &lexical exception
#\alarmx        ⟹ &lexical exception
#\alarm x       ⟹ U+0007
                ; followed by  x
#\Alarm         ⟹ &lexical exception
#\alert         ⟹ &lexical exception
#\xA            ⟹ U+000A
#\xFF           ⟹ U+00FF
#\xff           ⟹ U+00FF
#\x ff          ⟹ U+0078
                ; followed by another datum,  ff
#\x(ff)         ⟹ U+0078
                ; followed by another datum,
                ; a parenthesized  ff
#\(x)           ⟹ &lexical exception
#\(x            ⟹ &lexical exception
#\((x)          ⟹ U+0028
                ; followed by another datum,
                ; parenthesized  x
#\x00110000     ⟹ &lexical exception
                ; out of range
#\x000000001    ⟹ &lexical exception
                ; too many digits
#\xD800         ⟹ &lexical exception
                ; in excluded range
```

(The notation `&lexical` *exception* means that the line in question is a lexical syntax violation.)

Case is significant in `#\`⟨character⟩, and in in `#\`⟨character name⟩, but not in `#\x`⟨digit 16⟩$^+$. The character after a ⟨character⟩ must be a delimiter character such as a space or parenthesis. This rule resolves various ambiguous cases, for example, the sequence of characters "`#\space`" could be taken to be either a representation of the space character or a representation of the character "`#\s`" followed by a representation of the symbol "`pace`."

3.2.6. Strings

String are written as sequences of characters enclosed within doublequotes (`"`). Within a string literal, various escape sequences denote characters other than themselves. Escape sequences always start with a backslash (\):

- `\a` : alarm, U+0007
- `\b` : backspace, U+0008
- `\t` : character tabulation, U+0009
- `\n` : linefeed, U+000A
- `\v` : line tabulation, U+000B
- `\f` : formfeed, U+000C
- `\r` : return, U+000D
- `\"` : doublequote, U+0022
- `\\` : backslash, U+005C
- `\`⟨linefeed⟩⟨intraline whitespace⟩ : nothing
- `\`⟨space⟩ : space, U+0020 (useful for terminating the previous escape sequence before continuing with whitespace)

- `\x`⟨digit 16⟩$^+$`;` : (note the terminating semi-colon) where no more than eight ⟨digit 16⟩s are provided, and the sequence of ⟨digit 16⟩s forms a hexadecimal number between 0 and `#x10FFFF` excluding the range [#xD800, #xDFFF].

These escape sequences are case-sensitive, except that ⟨digit 16⟩ can be an uppercase or lowercase hexadecimal digit.

Any other character in a string after a backslash is an error. Any character outside of an escape sequence and not a doublequote stands for itself in the string literal. For example the single-character string `"λ"` (double quote, a lower case lambda, double quote) denotes the same string literal as `"\x03bb;"`.

Examples:

```
"abc"          ⟹  U+0061, U+0062, U+0063
"\x41;bc"      ⟹  "Abc" ; U+0041, U+0062, U+0063
"\x41; bc"     ⟹ "A bc"
                ; U+0041, U+0020, U+0062, U+0063
"\x41bc;"      ⟹  U+41BC
"\x41"         ⟹ &lexical exception
"\x;"          ⟹ &lexical exception
"\x41bx;"      ⟹ &lexical exception
"\x00000041;"  ⟹  "A" ; U+0041
"\x0010FFFF;"  ⟹ U+10FFFF
"\x00110000;"  ⟹  &lexical exception
                ; out of range
"\x000000001;"⟹ &lexical exception
                ; too many digits
"\xD800;"      ⟹ &lexical exception
                ; in excluded range
```

3.2.7. Numbers

The syntax of written representations for numbers is described formally by the ⟨number⟩ rule in the formal grammar. Note that case is not significant in numerical constants.

A number may be written in binary, octal, decimal, or hexadecimal by the use of a radix prefix. The radix prefixes are `#b` (binary), `#o` (octal), `#d` (decimal), and `#x` (hexadecimal). With no radix prefix, a number is assumed to be expressed in decimal.

A numerical constant may be specified to be either exact or inexact by a prefix. The prefixes are `#e` for exact, and `#i` for inexact. An exactness prefix may appear before or after any radix prefix that is used. If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a "`#`" character in the place of a digit; otherwise it is exact.

In systems with inexact numbers of varying precisions, it may be useful to specify the precision of a constant. For this purpose, numerical constants may be written with an exponent marker that indicates the desired precision of the inexact representation. The letters `s`, `f`, `d`, and `l` specify the use of *short*, *single*, *double*, and *long* precision, respectively. (When fewer than four internal inexact representations exist, the four size specifications are mapped onto those available. For example, an implementation with two internal representations may map short and single together and long and double together.) In addition, the exponent marker `e` specifies the default precision for the implementation. The default precision has at least as much precision as *double*, but implementations may wish to allow this default to be set by the user.

```
3.14159265358979F0
        Round to single — 3.141593
0.6L0
        Extend to long — .600000000000000
```

If x is an external representation of an inexact real number that contains no vertical bar, and p is a sequence of 1 or more decimal digits, then x`|`p is an external representation that denotes the best binary floating point approximation to x using a p-bit significand. For example, `1.1|53` is an external representation for the best approximation to 1.1 in IEEE double precision.

If x is an external representation of an inexact real number that contains no vertical bar, then x by itself should be regarded as equivalent to x`|53`.

Implementations that use binary floating point representations of real numbers should represent x`|`p using a p-bit significand if practical, or by a greater precision if a p-bit significand is not practical, or by the largest available precision if p or more bits of significand are not practical within the implementation.

Note: The precision of a significand should not be confused with the number of bits used to represent the significand. In the IEEE floating point standards, for example, the significand's most significant bit is implicit in single and double precision but is explicit in extended precision. Whether that bit is implicit or explicit does not affect the mathematical precision. In implementations that use binary floating point, the default precision can be calculated by calling the following procedure:

```
(define (precision)
  (do ((n 0 (+ n 1))
       (x 1.0 (/ x 2.0)))
    ((= 1.0 (+ 1.0 x)) n)))
```

Note: When the underlying floating-point representation is IEEE double precision, the `|`p suffix should not always be omitted: Denormalized numbers have diminished precision, and therefore should carry a `|`p suffix with the actual width of the significand.

The literals `+inf.0` and `-inf.0` represent positive and negative infinity, respectively. The `+nan.0` literal represents the NaN that is the result of `(/ 0.0 0.0)`, and may represent other NaNs as well.

If a ⟨decimal 10⟩ contains no vertical bar and does not contain one of the exponent markers `s`, `f`, `d`, or `l`, but does contain a decimal point or the exponent marker `e`, then it is an external representation for a flonum. Furthermore `inf.0`, `+inf.0`, `-inf.0`, `nan.0`, `+nan.0`, and `-nan.0` are external representations for flonums. Some or all of the other external representations for inexact reals may also represent flonums, but that is not required by this report.

If a ⟨decimal 10⟩ contains a non-empty ⟨mantissa width⟩ or one of the exponent markers `s`, `f`, `d`, or `l`, then it represents an inexact number, but does not necessarily represent a flonum.

3.3. Read syntax

The read syntax describes the syntax of syntactic datums in terms of a sequence of ⟨lexeme⟩s, as defined in the lexical syntax.

Syntactic datums include the lexeme datums described in the previous section as well as the following constructs for forming compound structure:

- pairs and lists, enclosed by `( )` or `[ ]` (see section 3.3.3)
- vectors (see section 3.3.2)

Note that the sequence of characters "`(+ 2 6)`" is *not* a syntactic datum representing the integer 8, even though it *is* a base-library expression evaluating to the integer 8; rather, it is a datum representing a three-element list, the elements of which are the symbol `+` and the integers 2 and 6.

3.3.1. Formal account

The following grammar describes the syntax of syntactic datums in terms of various kinds of lexemes defined in the grammar in section 3.2:

⟨datum⟩ ⟶ ⟨simple datum⟩
  | ⟨compound datum⟩
⟨simple datum⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
  | ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩
⟨symbol⟩ ⟶ ⟨identifier⟩
⟨compound datum⟩ ⟶ ⟨list⟩ | ⟨vector⟩
⟨list⟩ ⟶ `(`⟨datum⟩*`)`
  | `[`⟨datum⟩*`]`
  | `(`⟨datum⟩$^+$ `.` ⟨datum⟩`)`
  | `[`⟨datum⟩$^+$ `.` ⟨datum⟩`]`
  | ⟨abbreviation⟩
⟨abbreviation⟩ ⟶ ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ ⟶ `'` | `` ` `` | `,` | `,@` | `#'` | `` #` `` | `#,` | `#,@`
⟨vector⟩ ⟶ `#(`⟨datum⟩*`)`
⟨bytes⟩ ⟶ `#vu8(`⟨u8⟩*`)`
⟨u8⟩ ⟶ ⟨any ⟨number⟩ denoting an exact integer in $\{0, \ldots, 255\}$⟩

3.3.2. Vectors

Vector datums, denoting vectors of values (see section 9.16, are written using the notation `#(`⟨datum⟩ `...)`. For example, a vector of length 3 containing the number zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be written as following:

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector, and is not a base-library expression that evaluates to a vector.

3.3.3. Pairs and lists

List and pair datums, denoting pairs and lists of values (see section 9.12) are written using parentheses or brackets. Matching pairs of parentheses that occur in the rules of ⟨list⟩ are equivalent to matching pairs of brackets.

The most general notation for Scheme pairs as syntactic datums is the "dotted" notation `(`⟨datum$_1$⟩ `.` ⟨datum$_2$⟩`)` where ⟨datum$_1$⟩ is the representation of the value of the car field and ⟨datum$_2$⟩ is the representation of the value of the cdr field. For example `(4 . 5)` is a pair whose car is 4 and whose cdr is 5. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()` . For example,

```
(a b c d e)
```

and

```
(a . (b . (c . (d . (e . ()))))))
```

are equivalent notations for a list of symbols.

The general rule is that, if a dot is followed by an open parenthesis, the dot, the open parenthesis, and the matching closing parenthesis can be omitted in the external representation.

3.3.4. Bytes objects

Bytes datums, denoting bytes objects (see section 11), are written using the notation `#vu8(`⟨u8⟩ ...`)`, where the ⟨u8⟩s repersent the octets of the bytes object. For example, a bytes object of length 3 containing the octets 2, 24, and 123 can be written as follows:

```
#vu8(2 24 123)
```

Note that this is the external representation of a bytes object, and is not an expression that evaluates to a bytes object.

3.3.5. Abbreviations

`'`⟨datum⟩
`` ` ``⟨datum⟩
`,`⟨datum⟩
`,@`⟨datum⟩
`#'`⟨datum⟩
`` #` ``⟨datum⟩
`#,`⟨datum⟩
`#,@`⟨datum⟩

Each of these is an abbreviation:
`'`⟨datum⟩ for `(quote` ⟨datum⟩`)`,
`` ` ``⟨datum⟩ for `(quasiquote` ⟨datum⟩`)`,
`,`⟨datum⟩ for `(unquote` ⟨datum⟩`)`,
`,@`⟨datum⟩ for `(unquote-splicing` ⟨datum⟩`)`,
`#'`⟨datum⟩ for `(syntax` ⟨datum⟩`)`,
`` #` ``⟨datum⟩ for `(quasisyntax` ⟨datum⟩`)`,
`#,`⟨datum⟩ for `(unsyntax` ⟨datum⟩`)`, and
`#,@`⟨datum⟩ for `(unsyntax-splicing` ⟨datum⟩`)`.

4. Semantic concepts

4.1. Programs and libraries

A Scheme program consists of a *script* together with a set of *libraries*, each of which defines a part of the program connected to the others through explicitly specified exports and imports. A library consists of a set of export and import specifications and a body, which consists of declarations, definitions, and expressions; a script is similar to a library, but has no export specifications. Chapters 6 and 7 describe the syntax and semantics of libraries and scripts, respectively. Subsequent chapters describe various standard libraries provided by a Scheme system. In particular, chapter 9 describes a base library that defines many of the constructs traditionally associated with Scheme programs.

The division between the base library and other standard libraries is based on use, not on construction. In particular, some facilities that are typically implemented as "primitives" by a compiler or run-time libraries rather than in terms of other standard procedures or syntactic forms are not part of the base library, but are defined in separate libraries. Examples include the fixnums and flonums libraries, the exceptions and conditions libraries, and the libraries for records.

4.2. Variables, syntactic keywords, and regions

In a library body, an identifier may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a type of syntax is called a *syntactic keyword* and is said to be *bound* to that syntax. An identifier that names a location is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new kinds of syntax and to bind syntactic keywords to those new syntaxes, while other expression types create new locations and bind variables to those locations. These expression types are called *binding constructs*. The constructs in the base library that bind syntactic keywords are listed in section 6.3.2. The most fundamental of the variable binding constructs is the `lambda` expression, because all other variable binding constructs can be explained in terms of `lambda` expressions. The other variable binding constructs are `let`, `let*`, `letrec*`, `letrec`, `let-values`, `let*-values`, `do`, and `case-lambda` expressions (see sections 9.5.2, 9.5.6, 9.19, and 20.2).

Like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp, Scheme is a statically scoped language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program text within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a `lambda` expression, for example, then its region is the entire `lambda` expression. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use. If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the variable in the top level environment of the library body or a binding imported from another library. (See chapter 6.) If there is no binding for the identifier, it is said to be *unbound*.

4.3. Exceptional situations

A variety of exceptional situations are distinguished in this report, among them violations of program syntax, violations of a procedure's specification, violations of implementation restrictions, and exceptional situations in the environment. When an exception is raised, an object is provided that describes the nature of the exceptional siutation. The report uses the condition system described in section 14.2 to describe exceptional situations, classifying them by condition types.

For most of the exceptional situations described in this report, portable programs cannot rely upon the exception being continuable at the place where the situation was detected. For those exceptions, the exception handler that is invoked by the exception should not return. In some cases, however, continuing is permissible; the handler may return. See section 14.1.

An *implementation restriction* is a limitation imposed by an implementation. Implementations are required to raise an exception when they are unable to continue correct execution of a correct program due to some implementation restriction.

Some possible implementation restrictions such as the lack of representations for NaNs and infinities (see section 16.1) are anticipated by this report, and implementations must raise an exception of the appropriate condition type if they encounter such a situation.

Implementation restrictions not explicitly covered in this report are of course discouraged, but implementations are required to report violations of implementation restrictions. For example, an implementation may raise an exception with condition type `&implementation-restriction` if it does not have enough storage to run a program.

The above requirements for violations and implementation restrictions apply only in scripts and libraries that are said to be *safe*. In *unsafe* code, implementations might not raise the exceptions that are normally raised in those situations. The distinction between safe and unsafe code is explained in section 4.4.

4.4. Safety

The standard libraries whose exports are described by this document are said to be *safe libraries*. Libraries and scripts that import only from safe libraries, and do not contain any `(safe 0)` or `unsafe` declarations (see section 9.22), are also said to be safe.

As defined by this document, the Scheme programming language is safe in the following sense: If a Scheme script is said to be safe, then its execution cannot go so badly wrong as to crash or to continue to execute while behaving in ways that are inconsistent with the semantics described in this document, unless said execution first encounters some implementation restriction or other defect in the implementation of Scheme that is executing the script.

Violations of an implementation restriction must raise an exception with condition type `&implementation-restriction`, as must all violations and errors that would otherwise threaten system integrity in ways that might result in execution that is inconsistent with the semantics described in this document.

The above safety properties are guaranteed only for scripts and libraries that are said to be safe. Implementations may provide access to unsafe libraries, and may interpret `(safe 0)` and `unsafe` declarations in ways that cannot guarantee safety.

4.5. Multiple return values

A Scheme expression can evaluate to an arbitrary finite number of values. These values are passed to the expression's continuation.

Not all continuations accept any number of values: A continuation that accepts the argument to a procedure call is guaranteed to accept exactly one value. The effect of passing some other number of values to such a continuation is unspecified. The `call-with-values` procedure described in section 9.18 makes it possible to create continuations that accept specified numbers of return values. If the number of return values passed to a continuation created by a call to `call-with-values` is not accepted by its consumer that was passed in that call, then an exception is raised.

A number of forms in the base library have sequences of expressions as subforms that are evaluated sequentially, with the return values of all but the last expression being discarded. The continuations discarding these values accept any number of values.

4.6. Storage model

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the `string-set!` procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as `car`, `vector-ref`, or `string-ref`, is equivalent in the sense of `eqv?` (section 9.6) to the object last stored in the location before the fetch.

Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use. Whenever this report speaks of storage being allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them.

In many systems it is desirable for constants (i.e. the values of literal expressions) to reside in read-only-memory. To express this, it is convenient to imagine that every object that denotes locations is associated with a flag telling whether that object is mutable or immutable. In such systems literal constants and the strings returned by `symbol->string` are immutable objects, while all objects created by the other procedures listed in this report are mutable. An attempt to store a new value into a location that is denoted by an immutable object should raise an exception.

4.7. Proper tail recursion

Implementations of Scheme are required to be *properly tail-recursive*. Procedure calls that occur in certain syntactic contexts defined below are 'tail calls'. A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls. A call is *active* if the called procedure may still return. Note that this includes calls that may be returned from either by the current continuation or by continuations captured earlier by `call-with-current-continuation` that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned. A formal definition of proper tail recursion can be found in [11]. The rules for identifying tail calls in base-library constructs are described in section 9.23.

Rationale:

Intuitively, no space is needed for an active tail call because the continuation that is used in the tail call has the same semantics as the continuation passed to the procedure containing the call. Although an improper implementation might use a new continuation in the call, a return to this new continuation would be followed immediately by a return to the continuation passed to the procedure. A properly tail-recursive implementation returns to that continuation directly.

Proper tail recursion was one of the central ideas in Steele and Sussman's original version of Scheme. Their first Scheme interpreter implemented both functions and actors. Control flow was expressed using actors, which differed from functions in that they passed their results on to another actor instead of returning to a caller. In the terminology of this section, each actor finished with a tail call to another actor.

Steele and Sussman later observed that in their interpreter the code for dealing with actors was identical to that for functions and thus there was no need to include both in the language.

5. Notation and terminology

5.1. Entry format

The chapters describing bindings in the base library and the standard libraries are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

template *category*

If *category* is "syntax", the entry describes a special syntactic form, and the template gives the syntax of the form. Even though the template is written in a notation similar to a right-hand side of the BNF rules in chapter 3, it describes the set of forms equivalent to the forms matching the template as syntactic datums.

Components of the form described by a template are designated by syntactic variables, which are written using angle brackets, for example, ⟨expression⟩, ⟨variable⟩. Case is insignificant in syntactic variables. Syntactic variables should be understood to denote other forms, or, in some cases, sequences of them. A syntactic variable may refer to a non-terminal in the grammar for syntactic datums, in which case only forms matching that non-terminal are permissible in that position. For example, ⟨expression⟩ stands for any form which is a syntactically valid expression. Other non-terminals that are used in templates will be defined as part of the specification.

The notation

⟨thing$_1$⟩ ...

indicates zero or more occurrences of a ⟨thing⟩, and

⟨thing$_1$⟩ ⟨thing$_2$⟩ ...

indicates one or more occurrences of a ⟨thing⟩.

It is a syntax violation if a component of a form does not have the shape specified by a template—an exception with condition type `&syntax` is raised at expansion time.

Descriptions of syntax may express other restrictions on the components of a form. Typically, such a restriction is formulated as a phrase of the form "⟨x⟩ must be a ..." (or otherwise using the word "must.") As with implicit restrictions, such a phrase means that an exception with condition type `&syntax` is raised if the component does not meet the restriction.

If *category* is "procedure", then the entry describes a procedure, and the header line gives a template for a call to the procedure. Parameter names in the template are *italicized*. Thus the header line

`(vector-ref` *vector k*`)` procedure

indicates that the built-in procedure `vector-ref` takes two arguments, a vector *vector* and an exact non-negative integer k (see below). The header lines

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

indicate that the `make-vector` procedure takes either one or two arguments. The parameter names are case-insensitive: *Vector* is the same as *vector*.

An operation that is presented with an argument that it is not specified to handle raises an exception with condition type `&contract`. Also, if the number of arguments presented to an operation does not match any specified count, an exception with condition type `&contract` must be raised.

For succinctness, we follow the convention that if a parameter name is also the name of a type, then the corresponding argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions imply type restrictions:

obj	any object
z	complex number
x	real number
y	real number
q	rational number
n	integer
k	exact non-negative integer
octet	exact integer in {0, ..., 255}
bytes	exact integer in {-128, ..., 127}
char	character (see section 9.14)
pair	pair (see section 9.12)
list	list (see section 5.2)
vector	vector (see section 9.16)
string	string (see section 9.15)
condition	condition (see section 14.2)
bytes	bytes object (see chapter 11)
proc	procedure (see section 1.5)

Other type restrictions are expressed through parameter naming conventions that are described in specific chapters. For example, chapter 16 uses a number of special parameter variables for the various subsets of the numbers.

Descriptions of procedures may express other restrictions on the arguments of a procedure. Typically, such a restriction is formulated as a phrase of the form "x must be a ..." (or otherwise using the word "must.") As with implicit restrictions, such a phrase means that an exception with condition type `&contract` is raised if the argument does not meet the restriction.

If *category* is something other than "syntax" and "procedure," then the entry describes a non-procedural value, and the *category* describes the type of that value. The header line

`&who` condition type

indicates that `&who` is a condition type.

The description of an entry occasionally states that it is *the same* as another entry. This means that both entries are equivalent. Specifically, it means that if both entries have the same name and are thus exported from different libraries, the entries from both libraries can be imported under the same name without conflict.

5.2. List arguments

List arguments are immutable in programs that do not make use of the `(r6rs mutable-pairs)` library. In such programs, a procedure accepting a list as an argument can check whether the argument is a list by traversing it.

In programs that mutate pairs through use of the `(r6rs mutable-pairs)` library, a pair that is the head of a list at one moment may not always be the head of a list. Thus a traversal of the structure cannot by itself guarantee that the structure is a list; one must also know that no concurrent or interleaved computation can mutate the pairs of the structure. This greatly complicates the description of how certain procedures must verify that their arguments are valid.

For that reason, the specifications of procedures that accept lists generally assume that those lists are not mutated. Section 23.2 relaxes that assumption and states more precise restrictions on the arguments to these procedures.

5.3. Evaluation examples

The symbol "$\Longrightarrow$" used in program examples should be read "evaluates to." For example,

```
(* 5 8)                    ⟹  40
```

means that the expression `(* 5 8)` evaluates to the object `40`. Or, more precisely: the expression given by the sequence of characters "`(* 5 8)`" evaluates, in the initial environment, to an object that may be represented externally by the sequence of characters "`40`". See section 3.3 for a discussion of external representations of objects.

The "$\Longrightarrow$" symbol is also used when the evaluation of an expression raises an exception. For example,

```
(integer->char #xD800)      ⟹ &contract exception
```

means that the evaluation of the expression `(integer->char #xD800)` causes an exception with condition type `&contract` to be raised.

5.4. Unspecified behavior

If the value of an expression is said to be "unspecified," then the expression must evaluate without raising an exception, but the values returned depends on the implementation; this report explicitly does not say what values should be returned.

Some expressions are specified to return *the* unspecified value, which is a special value returned by the `unspecified` procedure. (See section 9.8.) In this case, the return value is meaningless, and programmers are discouraged from relying on its specific nature.

5.5. Exceptional situations

When speaking of an exceptional situation (see section 4.3), this report uses the phrase "an exception is raised" to indicate that implementations must detect the situation and report it to the program through the exception system described in chapter 14.

Several variations on "an exception is raised" are possible:

- "An exception should be raised" means that implementations are encouraged, but not required, to detect the situation and to raise an exception.
- "An exception may be raised" means that implementations are allowed, but not required or encouraged, to detect the situation and to raise an exception.
- "An exception might be raised" means that implementations are allowed, but discouraged, to detect the situation and to raise an exception.

This report uses the phrase "an exception with condition type t" to indicate that the object provided with the exception is a condition object of the specified type.

The phrase "a continuable exception is raised" indicates an exceptional situation that permits the exception handler to return, thereby allowing program execution to continue at the place where the original exception occurred. See sectionj 14.1.

For example, an exception with condition type `&contract` is raised if a procedure is passed an argument that the procedure is not explicitly specified to handle, even though such domain exceptions are not always mentioned in this report.

5.6. Naming conventions

By convention, the names of procedures that always return a boolean value usually end in "`?`". Such procedures are called predicates.

By convention, the names of procedures that store values into previously allocated locations (see section 4.6) usually end in "`!`". Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is the unspecified value (see section 9.8), but this convention is not always followed.

By convention, "`->`" appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, `list->vector` takes a list and returns a vector whose elements are the same as those of the list.

5.7. Syntax violations

Scheme implementations conformant with this report must detect violations of the syntax. A *syntax violation* is an error with respect to the syntax of library bodies, script bodies, or the "syntax" entries in the specification of the base library or the standard libraries. Moreover, attempting to assign to an immutable variable (i.e., the variables exported by a library; see section 6.1) is also considered a syntax violation.

If a script or library form is not syntactically correct, then the execution of that script or library must not be allowed to begin.

6. Libraries

The library system presented here is designed to let programmers share libraries, i.e., code that is intended to be incorporated into larger programs, and especially into programs that use library code from multiple sources. The library system supports macro definitions within libraries, allows macro exports, and distinguishes the phases in which definitions and imports are needed. This chapter defines the notation for libraries and a semantics for library expansion and execution.

Libraries address the following specific goals:

- Separate compilation and analysis; no two libraries have to be compiled at the same time (i.e., the meanings of two libraries cannot depend on each other cyclically, and compilation of two different libraries cannot rely on state shared across compilations), and significant program analysis can be performed without examining a whole program.
- Independent compilation/analysis of unrelated libraries, where "unrelated" means that neither depends on the other through a transitive closure of imports.
- Explicit declaration of dependencies, so that the meaning of each identifier is clear at compile time, and

so that there is no ambiguity about whether a library needs to be executed for another library's compile time and/or run time.

- Namespace management, so that different library producers are unlikely to define the same top-level name.

It does not address the following:

- Mutually dependent libraries.
- Separation of library interface from library implementation.
- Code outside of a library (e.g., `5` by itself as a program).
- Local modules and local imports.

6.1. Library form

A library declaration contains the following elements:

- a name for the library (possibly compound, with versioning),
- a list of exports, which name a subset of the bindings defined within or imported into the library,
- a list of import dependencies, where each dependency specifies:
  - the imported library's name,
  - the relevant phases, e.g., expand or run time, and
  - the subset of the library's exports to make available within the importing library, and the local names to use within the importing library for each of the library's exports, and
- a library body, consisting of a sequence of definitions preceded by a sequence of declarations and followed by a sequence of expressions.

A library definition must have the following form:

```
(library ⟨library name⟩
  (export ⟨export spec⟩ ...)
  (import ⟨import spec⟩ ...)
  ⟨library body⟩)
```

The ⟨library name⟩ specifies the name of the library, the `export` form specifies the exported bindings, and the `import` form specifies the imported bindings. The ⟨library body⟩ specifies the set of definitions, both for local (unexported) and exported bindings, and the set of initialization expressions (commands) to be evaluated for their effects. The exported bindings may be defined within the library or imported into the library.

An identifier can be imported from two or more libraries or for two phases from the same library only if the binding exported by each library is the same (i.e., the binding is defined in one library, and it arrives through the imports only by exporting and re-exporting). Otherwise, no identifier can be imported multiple times, defined multiple times, or both defined and imported. No identifiers are visible within a library except for those explicitly imported into the library or defined within the library.

A ⟨library name⟩ must be one of the following:

```
⟨identifier⟩
(⟨identifier₁⟩ ⟨identifier₂⟩ ... ⟨version⟩)
```

where ⟨version⟩ is empty or has the following form:

```
(⟨subversion₁⟩ ⟨subversion₂⟩ ...)
```

Each ⟨subversion⟩ must be an exact nonnegative integer.

As a ⟨library name⟩, ⟨identifier⟩ is shorthand for `(`⟨identifier⟩`)`.

Each ⟨import spec⟩ specifies a set of bindings to be imported into the library, the phases in which they are to be available, and the local names by which they are to be known. A ⟨import spec⟩ must be one of the following:

```
⟨import set⟩
(for ⟨import set⟩ ⟨import phase⟩ ...)
```

An ⟨import phase⟩ is one of the following:

```
run
expand
(meta ⟨level⟩)
```

where ⟨level⟩ is an exact nonnegative integer.

As an ⟨import phase⟩, `run` is an abbreviation for `(meta 0)`, and `expand` is an abbreviation for `(meta 1)`. Phases are discussed in section 6.2.

An ⟨import set⟩ names a set of bindings from another library, and possibly specifies local names for the imported bindings. It must be one of the following:

```
⟨library reference⟩
(only ⟨import set⟩ ⟨identifier⟩ ...)
(except ⟨import set⟩ ⟨identifier⟩ ...)
(add-prefix ⟨import set⟩ ⟨identifier⟩)
(rename ⟨import set⟩ (⟨identifier⟩ ⟨identifier⟩) ...)
```

A ⟨library reference⟩ identifies a library by its (possibly compound) name and optionally by its version. It must have one the following forms:

```
⟨identifier⟩
(⟨identifier₁⟩ ⟨identifier₂⟩ ... ⟨version reference⟩)
```

⟨Identifier⟩ is shorthand for (⟨identifier⟩). A ⟨version reference⟩ must have one of the following forms:

⟨empty⟩
(⟨subversion reference$_1$⟩ ⟨subversion reference$_2$⟩ ...)

A ⟨subversion reference⟩ must have one of the following forms:

⟨subversion⟩
⟨subversion condition⟩

where a ⟨subversion condition⟩ must have one of these forms:

(>= ⟨subversion⟩)
(<= ⟨subversion⟩)
(and ⟨subversion condition$_1$⟩ ⟨subversion condition$_2$⟩ ...)
(or ⟨subversion condition$_1$⟩ ⟨subversion condition$_2$⟩ ...)
(not ⟨subversion condition⟩)

The sequence of identifiers in the importing library's ⟨library reference⟩ must match the sequence of identifiers in the imported library's ⟨library name⟩. The importing library's ⟨version reference⟩ specifies a predicate on a prefix of the imported library's ⟨version⟩. Each integer must match exactly and each condition has the expected meaning. Everything beyond the prefix specified in the version reference matches unconditionally. When more than one library is identified by a library reference, the choice of libraries is determined in some implementation-dependent manner.

To avoid problems such as incompatible types and replicated state, two libraries whose library names contain the same sequence of identifiers but whose versions do not match cannot co-exist in the same program.

By default, all of an imported library's exported bindings are made visible within an importing library using the names given to the bindings by the imported library. The precise set of bindings to be imported and the names of those bindings can be adjusted with the `only`, `except`, `add-prefix`, and `rename` forms as described below.

- The `only` form produces a subset of the bindings from another ⟨import set⟩, including only the listed ⟨identifier⟩s; if any of the included ⟨identifier⟩s is not in ⟨import set⟩, an exception is raised.
- The `except` form produces a subset of the bindings from another ⟨import set⟩, including all but the listed ⟨identifier⟩s; if any of the excluded ⟨identifier⟩s is not in ⟨import set⟩, an exception is raised.
- The `add-prefix` adds the ⟨identifier⟩ prefix to each name from another ⟨import set⟩.
- The `rename` form, for each pair of identifiers (⟨identifier⟩ ⟨identifier⟩), removes a binding from the set from ⟨import set⟩, and adds it back with a different name. The first identifier is the original name, and the second identifier is the new name. If the original name is not in ⟨import set⟩, or if the new name is already in ⟨import set⟩, an exception is raised.

An ⟨export spec⟩ names a set of imported and locally defined bindings to be exported, possibly with different external names. An ⟨export spec⟩ must have one of the following forms:

⟨identifier⟩
(rename (⟨identifier⟩ ⟨identifier⟩) ...)

In an ⟨export spec⟩, an ⟨identifier⟩ names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A `rename` spec exports the binding named by the first ⟨identifier⟩ in each pair, using the second ⟨identifier⟩ as the external name.

The ⟨library body⟩ of a `library` form consists of forms that are classified into *declarations*, *definitions*, and *expressions*. Which forms belong to which class depends on the imported libraries and the result of expansion—see chapter 8. Generally, forms that are not declarations (see section 9.22 for declarations available through the base library) or definitions (see section 9.2 for definitions available through the base library) are expressions.

A ⟨library body⟩ is like a ⟨body⟩ (see section 9.4) except that ⟨library body⟩s need not include any expressions. It must have the following form:

⟨declaration⟩ ... ⟨definition⟩ ... ⟨expression⟩ ...

When base-library `begin` forms occur in a library body prior to the first expression, they are spliced into the body; see section 9.5.7. Some or all of the library body, including portions wrapped in `begin` forms, may be specified by a syntactic abstraction (see section 6.3.2).

The transformer expressions and transformer bindings are created from left to right, as described in chapter 8. The variable-definition right-hand-side expressions are evaluated from left to right, as if in an implicit `letrec*`, and the body expressions are also evaluated from left to right after the variable-definition right-hand-side expressions. A fresh location is created for each exported variable and initialized to the value of its local counterpart. The effect of returning twice to the continuation of the last body expression is unspecified.

The names `library`, `export`, `import`, `for`, `run`, `expand`, `meta`, `import`, `export`, `only`, `except`, and `rename` appearing in the library syntax are part of the syntax and are not reserved, i.e, the same can be used for other purposes within the library or even exported from or imported into a library with different meanings, without affecting their use in the `library` form.

In the case of any ambiguities that arise from the use of one of these names as a shorthand (single-identifier) library name, the ambiguity should be resolved in favor of the interpretation of the name as library syntax. For example, `(import (for lib expand))` should be taken

as importing library `lib` for `expand`, not as importing a library named `(for lib expand)`. The user can always eliminate such ambiguities by avoiding the shorthand ⟨library reference⟩ syntax when such an ambiguity might arise.

Bindings defined with a library are not visible in code outside of the library, unless the bindings are explicitly exported from the library. An exported macro may, however, *implicitly export* an otherwise unexported identifier defined within or imported into the library. That is, it may insert a reference to that identifier into the output code it produces.

All explicitly exported variables are immutable in both the exporting and importing libraries. An exception with condition type `&syntax` is thus raised if an explicitly exported variable appears on the left-hand side of a `set!` expression, either in the exporting or importing libraries. All other variables defined within a library are mutable.

All implicitly exported variables are also immutable in both the exporting and importing libraries. An exception with condition type `&syntax` is thus raised if a variable appears on the left-hand side of a `set!` expression in any code produced by an exported macro outside of the library in which the variable is defined. An exception with condition type `&syntax` is also raised if a reference to an assigned variable appears in any code produced by an exported macro outside of the library in which the variable is defined, where an assigned variable is one that appears on the left-hand side of a `set!` expression in the exporting library.

Note: The asymmetry in the exception-raising requirements for attempts to assign explicitly and implicitly exported variables reflects the fact that the error can be determined for implicitly exported variables only when the importing library is expanded.

6.2. Import and export phases

All bindings imported via a library's `import` form are *visible* throughout the library's ⟨library body⟩. An exception may be raised, however, if a binding is used out of its declared phase(s):

- Bindings used in run-time code must be imported "for `run`," which is equivalent to "for `(meta 0)`."
- Bindings used in the body of a transformer (appearing on the right-hand-side of a transformer binding) in run-time code must be imported "for `expand`," which is equivalent to "for `(meta 1)`,"
- Bindings used in the body of a transformer appearing within the body of a transformer in run-time code must be imported "for `(meta 2)`," and so on.

The effective import phases of an imported binding are determined by the enclosing `for` form, if any, in the `import` form of the importing library, in addition to the phase of the identifier in the exporting library. An ⟨import set⟩ without an enclosing `for` is equivalent to `(for` ⟨import set⟩ `run)`. Import and export phases are combined by pairwise addition of all phase combinations. For example, references to an imported identifier exported for phases p_a and p_b and imported for phases q_a, q_b, and q_c are valid at phases $p_a + q_q$, $p_a + q_b$, $p_a + q_c$, $p_b + q_q$, $p_b + q_b$, and $p_b + q_c$.

The export phase of an exported binding is `run` for all bindings that are defined within the exporting library. The export phases of a reexported binding, i.e., an export imported from another library, are the same as the effective import phases of that binding within the reexporting library.

The export phase of all bindings exported by the libraries defined in this report, except for the composite `r6rs` library (see chapter 21), is `run`, while the export phases for all bindings exported by `r6rs` are `run` and `expand`.

Rationale: The `r6rs` library is intended as a convenient import for libraries where fine control over imported bindings is not necessary or desirable. The `r6rs` library exports all bindings for `expand` as well as `run` so that it is convenient for writing macros as well as run-time code.

The effective import phases implicitly determine when information about a library must be available and also when the various forms contained within a library must be evaluated.

Every library can be characterized by expand-time information (minimally, its imported libraries, a list of the exported keywords, a list of the exported variables, and code to evaluate the transformer expressions) and run-time information (minimally, code to evaluate the variable definition right-hand-side expressions, and code to evaluate the body expressions). The expand-time information must be available to expand references to any exported binding, and the run-time information must be available to evaluate references to any exported variable binding.

If any of a library's bindings are imported by another library "for `expand`" (or for any meta level greater than 0), both expand-time and run-time information for the first library is made available when the second library is expanded. If any of a library's bindings are imported by another library "for `run`," the expand-time information for the first library is made available when the second library is expanded, and the run-time information for the first library is made available when the run-time information for the second library is made available.

It is also relevant when the code to evaluate a library's transformer expressions is executed and when the code to evaluate the library's variable-definition right-hand-side

expressions and body expressions is executed. Executing the transformer expressions is also said to be *visiting* the library and to executing the variable-definition right-hand-side expressions and body expressions as *invoking* the library. A library must be visited before code that uses its bindings can be expanded, and it must be invoked before code that uses its bindings can be executed. Visiting or invoking a library may also trigger the visiting or invoking of other libraries.

More precisely, visiting a library at phase N causes the system to:

- Visit at phase N any library that is imported by this library "for `run`" and that is not yet visited at phase N.
- Visit at phase $N + M$ any library that is imported by this library "for `(meta` M`)`," $M > 0$ and that is not yet visited at phase $N + M$.
- Invoke at phase $N + M$ any library that is imported by this library "for `(meta` M`)`," $M > 0$ and that is not yet invoked at phase $N + M$.
- Evaluate the library's transformer expressions.

The order in which imported libraries are visited and invoked is not defined, but imported libraries must be visited and invoked before the library's transformer expressions are evaluated.

Similarly, invoking a library at meta phase N causes the system to:

- Invoke at phase N any library that is imported by this library "for `run`" and that is not yet invoked at phase N.
- Evaluate the library's variable-definition right-hand-side and body expressions.

The order in which imported libraries are invoked is not defined, but imported libraries must be invoked before the library's variable-definition right-hand-side and body expressions are evaluated.

An implementation is allowed to distinguish visits of a library across different phases or to treat a visit at any phase as a visit at all phases. Similarly, an implementation is allowed to distinguish invocations of a library across different phases or to treat an invocation at any phase as an invocation at all phases. An implementation is further allowed to start each expansion of a `library` form by removing all library bindings above phase 0. Thus, a portable library's meaning must not depend on whether the invocations are distinguished or preserved across phases or `library` expansions.

6.3. Primitive syntax

After the `import` form within a `library` form, the forms that constitute a library body depend on the libraries that are imported. In particular, imported syntactic keywords determine most of the available forms, and whether each form is a declaration, definition, or expression. A few form types are always available independent of imported libraries, however, including constant literals, variable references, procedure calls, and macro uses.

6.3.1. Primitive expression types

The entries in this section all describe expressions, which may occur in the place of ⟨expression⟩ syntactic variables. See also section 9.5

Constant literals

⟨constant⟩ syntax

Numerical constants, string constants, character constants, and boolean constants evaluate "to themselves."

```
"abc"                ⟹  "abc"
145932               ⟹  145932
#t                   ⟹  #t
```

As noted in section 4.6, the value of a literal expression may be immutable.

Variable references

⟨variable⟩ syntax

An expression consisting of a variable (section 4.2) is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is a syntax violation to reference an unbound variable.

```
; these examples assume the base library
; has been imported
(define x 28)
x                    ⟹  28
```

Procedure calls

`(`⟨operator⟩ ⟨operand$_1$⟩ `...)` syntax

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. A form in an expression context is a procedure call if ⟨operator⟩ is not an identifier bound as a syntactic keyword.

When an procedure call is evaluated, the operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments..

```
; these examples assume the base library
; has been imported
(+ 3 4)                    ⟹  7
((if #f + *) 3 4)          ⟹  12
```

If the value of ⟨operator⟩ is not a procedure, an exception with condition type `&contract` is raised.

Note: In contrast to other dialects of Lisp, the order of evaluation is unspecified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

Note: Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.

Note: In many dialects of Lisp, the form `()` is a legitimate expression. In Scheme, expressions written as list/pair forms must have at least one subexpression, so `()` is not a syntactically valid expression.

6.3.2. Macros

Scheme programs can define and use new derived expressions, definitions, and declarations, called *syntactic abstractions* or *macros*. A syntactic abstraction is created by binding a keyword to a *macro transformer* or, simply, *transformer*. . The transformer determines how a use of the macro is transcribed into a more primitive forms.

Macro uses typically have the form:

(⟨keyword⟩ ⟨datum⟩ ...)

where ⟨keyword⟩ is an identifier that uniquely determines the type of form. This identifier is called the *syntactic keyword*, or simply *keyword*, of the macro. The number of ⟨datum⟩s and the syntax of each depends on the syntactic abstraction. Macro uses can also take the form of improper lists, singleton identifiers, or `set!` forms, where the second subform of the `set!` is the keyword (see section 17.3:

(⟨keyword⟩ ⟨datum⟩ ... . ⟨datum⟩)
⟨keyword⟩
(`set!` ⟨keyword⟩ ⟨datum⟩)

The macro definition facility consists of two parts:

- A set of forms (`define-syntax`, section 9.3, `let-syntax` and `letrec-syntax`; see section 9.20) used to create bindings for keywords, associate them with macro transformers, and control the scope within which they are visible, and
- a facility (`syntax-case`; see chapter 17) for creating transformers via a pattern language that permits the use of arbitrary Scheme code, and a derived facility (`syntax-rules`; see section 9.21) for creating transformers via the pattern language only.

Keywords occupy the same name space as variables. That is, within the same scope, an identifier can be bound as a variable or keyword, or neither, but not both, and local bindings of either kind may shadow other bindings of either kind.

Macros defined using `syntax-rules` are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping [31, 32, 3, 10, 14]:

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier is in effect renamed throughout its scope to avoid conflicts with other identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

Macros defined using the `syntax-case` facility are also hygienic unless `datum->syntax` (see section 17.6) is used.

6.4. Examples

Examples for various ⟨import spec⟩s and ⟨export spec⟩s:

```
(library stack
  (export make push! pop! empty!)
  (import r6rs)

  (define (make) (list '()))
  (define (push! s v) (set-car! s (cons v (car s))))
  (define (pop! s) (let ([v (caar s)])
                     (set-car! s (cdar s))
                     v))
  (define (empty! s) (set-car! s '())))

(library balloons
  (export make push pop)
  (import r6rs)

  (define (make w h) (cons w h))
  (define (push b amt)
    (cons (- (car b) amt) (+ (cdr b) amt)))
  (define (pop b) (display "Boom! ")
                  (display (* (car b) (cdr b)))
                  (newline)))

(library party
  ;; Total exports:
  ;; make, push, push!, make-party, pop!
  (export (rename (balloon:make make)
                  (balloon:push push))
          push!
          make-party
          (rename (party-pop! pop!)))
```

```
(import r6rs
        (only stack make push! pop!) ; not empty!
        (add-prefix balloons balloon:))

;; Creates a party as a stack of balloons,
;; starting with two balloons
(define (make-party)
  (let ([s (make)]) ; from stack
    (push! s (balloon:make 10 10))
    (push! s (balloon:make 12 9))
    s))
(define (party-pop! p)
  (balloon:pop (pop! p))))


(library main
  (export)
  (import r6rs party)

  (define p (make-party))
  (pop! p)        ; displays "Boom! 108"
  (push! p (push (make 5 5) 1))
  (pop! p))       ; displays "Boom! 24"
```

Examples for macros and phases:

```
(library (my-helpers id-stuff)
  (export find-dup)
  (import r6rs)

  (define (find-dup l)
    (and (pair? l)
         (let loop ((rest (cdr l)))
           (cond
            [(null? rest) (find-dup (cdr l))]
            [(bound-identifier=? (car l) (car rest))
                                 (car rest)]
            [else (loop (cdr rest))])))))

(library (my-helpers values-stuff)
  (export mvlet)
  (import r6rs (for (my-helpers id-stuff) expand))

  (define-syntax mvlet
    (lambda (stx)
      (syntax-case stx ()
        [(_ [(id ...) expr] body0 body ...)
         (not (find-dup
                (syntax-object->list
                  (syntax (id ...)))))
         (syntax
           (call-with-values
               (lambda () expr)
             (lambda (id ...) body0 body ...)))]))))

(library let-div
  (export let-div)
  (import r6rs (my-helpers values-stuff))

  (define (quotient+remainder n d)
    (let ([q (quotient n d)])
      (values q (- n (* q d)))))
  (define-syntax let-div
    (syntax-rules ()
     [(_ n d (q r) body0 body ...)
      (mvlet [(q r) (quotient+remainder n d)]
        body0 body ...)])))
```

7. Scripts

A *script* specifies an entry point for defining and running a Scheme program. A script specifies a set of libraries to import and code to run. Through the imported libraries, whether directly or the transitive closure of importing, the script defines a complete Scheme program.

Scripts follow the convention of many common platforms of accepting a list of string *command-line arguments* that may be used to pass data to the script. Moreover, a script can return an exact integer specifying the script's *exit value*.

7.1. Script syntax

A script is a delimited piece of text, typically a file, that follows the following syntax:

⟨script⟩ ⟶ ⟨script header⟩ ⟨script substance⟩
⟨script header⟩ ⟶ `#!` ⟨space⟩ `/usr/bin/env` ⟨space⟩
    `scheme-script` ⟨linefeed⟩
⟨script substance⟩ ⟶ `#!r6rs` ⟨import form⟩ ⟨script body⟩
    | ⟨import form⟩ ⟨script body⟩
⟨import form⟩ ⟶ `(import` ⟨import spec⟩*`)`
⟨script body⟩ ⟶ ⟨script body form⟩* ⟨expression⟩
⟨script body form⟩ ⟶ ⟨declaration⟩
    | ⟨definition⟩
    | ⟨expression⟩

7.1.1. Script header

The first line of a script is:

```
#! /usr/bin/env scheme-script
```

Implementations are required to ignore the first line of a script, however, even if it is not the above. This allows script headers to be customized locally by altering the script header from its default portable form.

Implementations should provide an executable program named `scheme-script` that is capable of executing scripts on platforms where this makes sense. On most Unix-like systems, due to the use of the `/usr/bin/env` trampoline, this program may itself be a shell script.

Most platforms require that scripts be marked as executable in some way, the details of which vary by platform and are beyond the scope of this report. Platforms

which do not support the Unix-like script header syntax may need to use other mechanisms, such as a registered filename extension, in order to associate a script with the `scheme-script` executable.

7.1.2. Script substance

The rules for ⟨script substance⟩ specify syntax at the form level.

The ⟨import form⟩ is identical to the import clause in libraries (see section 6.1), and specifies a set of libraries to import. A ⟨script body⟩ is like a ⟨library body⟩ (see section 6.1), except that declarations, definitions and expressions may occur in any order, and that the final form of the script body must be an expression. Thus, the syntax specified by ⟨script body form⟩ refers to the result of macro expansion.

When base-library `begin` forms occur anywhere within a script body, they are spliced into the body; see section 9.5.7. Some or all of the script body, including portions wrapped in `begin` forms, may be specified by a syntactic abstraction (see section 6.3.2).

7.2. Script semantics

A script is executed by treating the script similarly to a library, and invoking it. The semantics of a script body may be roughly explained by a simple translation into a library body: All declarations at the script top level are moved to the front, and each ⟨expression⟩ that appears before a variable definition in the script body is converted into a dummy definition `(define` ⟨variable⟩ ⟨expression⟩`)`, where ⟨variable⟩ is fresh identifier. (It is generally impossible to determine which forms are declarations, definitions, and expressions without concurrently expanding the body, so the actual translation is somewhat more complicated; see chapter 8.)

A script may access its command-line arguments by calling the `command-line-arguments` procedure (see section 20.4). The final expression of a script must return an exact integer, which becomes the exit value of the script. How that exit value is communicated to the environment is implementation-specific. When a script is invoked as a Unix or Windows program, the exit value simply becomes the exit status of the program.

If an exception with a `&serious` condition is raised during the execution of the script, the default exception handler behaves as described in section 14.1, and an implementation-specific exit value is communicated to the environment. On Unix, this value is according to the definition of `EX_SOFTWARE` in the `sysexits.h` header [24].

8. Expansion process

Macro uses (see section 6.3.2) are expanded into *core forms* at the start of evaluation (before compilation or interpretation) by a syntax *expander*. (The set of core forms is implementation-dependent, as is the representation of these forms in the expander's output.) If the expander encounters a syntactic abstraction, it invokes the associated transformer to expand the syntactic abstraction, then repeats the expansion process for the form returned by the transformer. If the expander encounters a core form, it recursively processes the subforms, if any, and reconstructs the form from the expanded subforms. Information about identifier bindings is maintained during expansion to enforce lexical scoping for variables and keywords.

To handle internal definitions, the expander processes the initial forms in a ⟨body⟩ (see section 9.4) or ⟨library body⟩ (see section 6.1) from left to right. How the expander processes each form encountered as it does so depends upon the kind of form.

macro use The expander invokes the associated transformer to transform the macro use, then recursively performs whichever of these actions are appropriate for the resulting form.

`declaration` **form** If none of the body forms processed so far is a definition, the declaration is handled in some implementation-dependent fashion. It is a syntax violation for a declaration to appear after a definition.

`define-syntax` **form** The expander expands and evaluates the right-hand-side expression and binds the keyword to the resulting transformer.

`define` **form** The expander records the fact that the defined identifier is a variable but defers expansion of the right-hand-side expression until after all of the definitions have been processed.

`begin` **form** The expander splices the subforms into the list of body forms it is processing. (See section 9.5.7.)

`let-syntax` **or** `letrec-syntax` **form** The expander splices the inner body forms into the list of (outer) body forms it is processing, arranging for the keywords bound by the `let-syntax` and `letrec-syntax` to be visible only in the inner body forms.

expression, i.e., nondefinition The expander completes the expansion of the deferred right-hand-side forms and the current and remaining expressions in the body, then constructs a residual `letrec*` form from the defined variables, expanded right-hand-side expressions, and expanded body expressions.

It is a syntax violation if the keyword that identifies one of the body forms as a definition (derived or core) is redefined by the same definition or a later definition in the same body. To detect this error, the expander records the identifying keyword for each macro use, `define-syntax` form, `define` form, `begin` form, `let-syntax` form, and `letrec-syntax` form it encounters while processing the definitions and checks each newly defined identifier (`define` or `define-syntax` left-hand side) against the recorded keywords, as with `bound-identifier=?` (section 17.5). For example, the following forms are syntax violations.

```
(let ()
  (define define 17)
  define)

(let-syntax ([def0 (syntax-rules ()
                     [(_ x) (define x 0)])])
  (let ()
    (def0 z)
    (define def0 '(def 0))
    (list z def0)))
```

Expansion of each variable definition right-hand side is deferred until after all of the definitions have been seen so that each keyword and variable reference within the right-hand side resolves to the local binding, if any.

Note that this algorithm does not directly reprocess any form. It requires a single left-to-right pass over the definitions followed by a single pass (in any order) over the body expressions and deferred right-hand sides.

For example, in

```
(lambda (x)
  (define-syntax defun
    (syntax-rules ()
      [(_ (x . a) e) (define x (lambda a e))]))
  (defun (even? n) (or (= n 0) (odd? (- n 1))))
  (define-syntax odd?
    (syntax-rules () [(_ n) (not (even? n))]))
  (odd? (if (odd? x) (* x x) x)))
```

The definition of `defun` is encountered first, and the keyword `defun` is associated with the transformer resulting from the expansion and evaluation of the corresponding right-hand side. A use of `defun` is encountered next and expands into a `define` form. Expansion of the right-hand side of this define form is deferred. The definition of `odd?` is next and results in the association of the keyword `odd?` with the transformer resulting from expanding and evaluating the corresponding right-hand side. A use of `odd?` appears next and is expanded; the resulting call to `not` is recognized as an expression because `not` is bound as a variable. At this point, the expander completes the expansion of the current expression (the `not` call) and the deferred right-hand side of the `even?` definition; the uses of `odd?` appearing in these expressions are expanded using the transformer associated with the keyword `odd?`. The final output is the equivalent of

```
(lambda (x)
  (letrec* ([even?
             (lambda (n)
               (or (= n 0)
                   (not (even? (- n 1)))))])
    (not (even? (if (not (even? x)) (* x x) x)))))
```

although the structure of the output is implementation dependent.

Because definitions and expressions can be interleaved in a ⟨script body⟩ (see chapter 7), the expander's processing of a ⟨script body⟩ is somewhat more complicated. It behaves as described above for a ⟨body⟩ or ⟨library body⟩ with the following exceptions. First, it treats declarations that appear after any definitions or expressions as if they appeared before all of the definitions and expressions. Second, when the expander finds a nondefinition, it defers its expansion and continues scanning for definitions. Once it reaches the end of set of forms, it processes the deferred right-hand-side and body expressions, then constructs a residual `letrec*` form from the defined variables, expanded right-hand-side expressions, and expanded body expressions. For each body expression that appears before a variable definition in the body, a dummy binding is created at the corresponding place within the set of `letrec*` bindings, with a fresh temporary variable on the left-hand side and the expression on the right-hand side, so that left-to-right evaluation order is preserved.

9. Base library

This chapter describes Scheme's base library, which exports many of the procedure and syntax bindings that are traditionally associated with Scheme.

Section 9.23 defines the rules that identify tail calls and tail contexts in base-library constructs.

9.1. Base types

No object satisfies more than one of the following predicates:

```
boolean?          pair?
symbol?           number?
char?             string?
vector?           procedure?
unspecified?      eof-object?
null?
```

These predicates define the base types *boolean*, *pair*, *symbol*, *number*, *char* (or *character*), *string*, *vector*, and *procedure*. Moreover, the empty list is a special object of its

own type, as are the unspecified value, and the end of file oject.

Although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section 9.11, all values count as true in such a test except for `#f`. This report uses the word "true" to refer to any Scheme value except `#f`, and the word "false" to refer to `#f`.

9.2. Definitions

The `define` forms described in this section are definitions for value bindings and may appear anywhere other definitions may appear. See section 6.1.

A ⟨definition⟩ must have one of the following forms:

- `(define` ⟨variable⟩ ⟨expression⟩`)` This binds ⟨variable⟩ to a new location before assigning the value of ⟨expression⟩ to it.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                    ⟹  6
(define first car)
(first '(1 2))              ⟹  1
```

- `(define` ⟨variable⟩`)`

  This form is equivalent to

  `(define` ⟨variable⟩ `(unspecified))`

- `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`

  ⟨Formals⟩ must be either a sequence of zero or more variables, or a sequence of one or more variables followed by a space-delimited period and another variable (as in a lambda expression, see section 9.5.2). This form is equivalent to

  `(define` ⟨variable⟩
  `(lambda (`⟨formals⟩`)` ⟨body⟩`)).`

- `(define (`⟨variable⟩ `.` ⟨formal⟩`)` ⟨body⟩`)`

  ⟨Formal⟩ must be a single variable. This form is equivalent to

  `(define` ⟨variable⟩
  `(lambda` ⟨formal⟩ ⟨body⟩`)).`

- A syntax definition, see section 9.3.

9.3. Syntax definitions

Syntax definitions are established with `define-syntax`. A `define-syntax` form is a ⟨definition⟩ and may appear anywhere other definitions may appear.

`(define-syntax` ⟨variable⟩ ⟨transformer spec⟩`)` syntax

This binds the keyword ⟨variable⟩ to a transformer specification specified by ⟨transformer spec⟩, which must either be a `syntax-rules` form (see section 9.21), or evaluate, at macro-expansion time, to a transformer. (Section 17.3).

Keyword bindings established by `define-syntax` are visible throughout the body in which they appear, except where shadowed by other bindings, and nowhere else, just like variable bindings established by `define`. All bindings established by a set of internal definitions, whether keyword or variable definitions, are visible within the definitions themselves. For example:

```
(let ()
  (define even?
    (lambda (x)
      (or (= x 0) (odd? (- x 1)))))
  (define-syntax odd?
    (syntax-rules ()
      ((odd?  x) (not (even? x)))))
  (even? 10))               ⟹ #t
```

An implication of the left-to-right processing order (section 8) is that one internal definition can affect whether a subsequent form is also a definition. For example, the expression

```
(let ()
  (define-syntax bind-to-zero
    (syntax-rules ()
      ((bind-to-zero id) (define id 0))))
  (bind-to-zero x)
  x)                        ⟹ 0
```

This behavior is irrespective of any binding for `bind-to-zero` that might appear outside of the `let` expression.

9.4. Bodies and sequences

The body ⟨body⟩ of a `lambda`, `let`, `let*`, `let-values`, `let*-values`, `letrec*`, `letrec` expression or that of a definition with a body has the following form:

⟨declaration⟩ ... ⟨definition⟩ ... ⟨sequence⟩

⟨Declaration⟩ is according to section 9.22.

⟨Sequence⟩ has the following form:

⟨expression$_1$⟩ ⟨expression$_2$⟩ ...

Definitions may occur after the declarations in a ⟨body⟩. Such definitions are known as *internal definitions* as opposed to library body definitions.

With `lambda`, `let`, `let*`, `let-values`, `let*-values`, `letrec*`, and `letrec`, the identifier defined by an internal definition is local to the ⟨body⟩. That is, the identifier is bound, and the region of the binding is the entire ⟨body⟩. For example,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))            ⟹  45
```

When base-library `begin` forms occur in a body prior to the first expression, they are spliced into the body; see section 9.5.7. Some or all of the body, including portions wrapped in `begin` forms, may be specified by a syntactic abstraction (see section 6.3.2).

An expanded ⟨body⟩ (see chapter 8) containing internal definitions can always be converted into a completely equivalent `letrec*` expression. For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
  (letrec* ((foo (lambda (y) (bar x y)))
            (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

9.5. Expressions

The entries in this section describe the expressions of the base language, which may occur in the position of the ⟨expression⟩ syntactic variable. The expressions also include constant literals, variable references and procedure calls as described in section 6.3.1.

9.5.1. Literal expressions

`(quote` ⟨datum⟩`)` syntax

Syntax: ⟨Datum⟩ should be a datum value. *Semantics:* `(quote` ⟨datum⟩`)` evaluates to the datum denoted by ⟨datum⟩. (See section 3.3.). This notation is used to include literal constants in Scheme code.

```
(quote a)                   ⟹  a
(quote #(a b c))            ⟹  #(a b c)
(quote (+ 1 2))             ⟹  (+ 1 2)
```

As noted in section 3.3.5, `(quote` ⟨datum⟩`)` may be abbreviated as `'`⟨datum⟩:

```
'"abc"                      ⟹  "abc"
'145932                     ⟹  145932
'a                          ⟹  a
'#(a b c)                   ⟹  #(a b c)
'()                         ⟹  ()
'(+ 1 2)                    ⟹  (+ 1 2)
'(quote a)                  ⟹  (quote a)
''a                         ⟹  (quote a)
```

As noted in section 4.6, the value of a literal expression may be immutable.

9.5.2. Procedures

`(lambda` ⟨formals⟩ ⟨body⟩`)` syntax

Syntax: ⟨Formals⟩ must be a formal arguments list as described below, and ⟨body⟩ must be according to section 9.4.

Semantics: A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression is evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the `lambda` expression was evaluated is be extended by binding the variables in the formal argument list to fresh locations, and the resulting actual argument values are stored in those locations. Then, the expressions in the body of the `lambda` expression (which may contain internal definitions and thus represent a `letrec*` form, see section 9.4) are evaluated sequentially in the extended environment. The result(s) of the last expression in the body is(are) returned as the result(s) of the procedure call.

```
(lambda (x) (+ x x))        ⟹  a procedure
((lambda (x) (+ x x)) 4)    ⟹  8

((lambda (x)
   (define (p y)
     (+ y 1))
   (+ (p x) x))
 5)                         ⟹  11

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)     ⟹  3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                    ⟹  10
```

⟨Formals⟩ must have one of the following forms:

- (⟨variable$_1$⟩ ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments are stored in the bindings of the corresponding variables.
- ⟨variable⟩: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the ⟨variable⟩.

- (⟨variable$_1$⟩ ... ⟨variable$_n$⟩ . ⟨variable$_{n+1}$⟩): If a space-delimited period precedes the last variable, then the procedure takes n or more arguments, where n is the number of formal arguments before the period (there must be at least one). The value stored in the binding of the last variable is a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

```
((lambda x x) 3 4 5 6)      ⟹  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                   ⟹  (5 6)
```

It is a syntax violation for a ⟨variable⟩ to appear more than once in ⟨formals⟩.

Each procedure created as the result of evaluating a `lambda` expression is (conceptually) tagged with a storage location, in order to make `eqv?` and `eq?` work on procedures (see section 9.6).

9.5.3. Conditionals

`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)` syntax
`(if` ⟨test⟩ ⟨consequent⟩`)` syntax

Syntax: ⟨Test⟩, ⟨consequent⟩, and ⟨alternate⟩ must be expressions.

Semantics: An `if` expression is evaluated as follows: first, ⟨test⟩ is evaluated. If it yields a true value (see section 9.11), then ⟨consequent⟩ is evaluated and its value(s) is(are) returned. Otherwise ⟨alternate⟩ is evaluated and its value(s) is(are) returned. If ⟨test⟩ yields a false value and no ⟨alternate⟩ is specified, then the result of the expression is the unspecified value.

```
(if (> 3 2) 'yes 'no)       ⟹  yes
(if (> 2 3) 'yes 'no)       ⟹  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                ⟹  1
(if #f #f)                  ⟹  the unspecified value
```

9.5.4. Assignments

`(set!` ⟨variable⟩ ⟨expression⟩`)` syntax

⟨Expression⟩ is evaluated, and the resulting value is stored in the location to which ⟨variable⟩ is bound. ⟨Variable⟩ must be bound either in some region enclosing the `set!` expression or at the top level of a library body. The result of the `set!` expression is the unspecified value.

```
(let ((x 2))
  (+ x 1)
  (set! x 4)
  (+ x 1))                  ⟹  5
```

It is a syntax violation if ⟨variable⟩ refers to an immutable binding.

9.5.5. Derived conditionals

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntax

Syntax: Each ⟨clause⟩ must be of the form

(⟨test⟩ ⟨expression$_1$⟩ ...)

where ⟨test⟩ is any expression. Alternatively, a ⟨clause⟩ may be of the form

(⟨test⟩ `=>` ⟨expression⟩)

The last ⟨clause⟩ may be an "else clause," which has the form

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)`.

Semantics: A `cond` expression is evaluated by evaluating the ⟨test⟩ expressions of successive ⟨clause⟩s in order until one of them evaluates to a true value (see section 9.11). When a ⟨test⟩ evaluates to a true value, then the remaining ⟨expression⟩s in its ⟨clause⟩ are evaluated in order, and the result(s) of the last ⟨expression⟩ in the ⟨clause⟩ is(are) returned as the result(s) of the entire `cond` expression. If the selected ⟨clause⟩ contains only the ⟨test⟩ and no ⟨expression⟩s, then the value of the ⟨test⟩ is returned as the result. If the selected ⟨clause⟩ uses the `=>` alternate form, then the ⟨expression⟩ is evaluated. Its value must be a procedure that accepts one argument; this procedure is then called on the value of the ⟨test⟩ and the value(s) returned by this procedure is(are) returned by the `cond` expression. If all ⟨test⟩s evaluate to false values, and there is no else clause, then the result of the conditional expression is the unspecified value; if there is an else clause, then its ⟨expression⟩s are evaluated, and the value(s) of the last one is(are) returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))      ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))        ⟹  equal
(cond ('(1 2 3) => cadr)
      (else #f))            ⟹  2
```

A sample definition of `cond` in terms of simpler forms is in appendix B.

`(case` ⟨key⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntax

Syntax: ⟨Key⟩ must be any expression. Each ⟨clause⟩ has one of the following forms:

((⟨datum$_1$⟩ ...) ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...)
`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)`

The second form, which specifies an "else clause," may only appear as the last ⟨clause⟩. Each ⟨datum⟩ is an external representation of some object. The datums denoted by the ⟨datum⟩s must not be distinct.

Semantics: A `case` expression is evaluated as follows. ⟨Key⟩ is evaluated and its result is compared against each

the datum denoted by each ⟨datum⟩. If the result of evaluating ⟨key⟩ is equivalent (in the sense of `eqv?`; see section 9.6) to a datum, then the expressions in the corresponding ⟨clause⟩ are evaluated from left to right and the result(s) of the last expression in the ⟨clause⟩ is(are) returned as the result(s) of the `case` expression. If the result of evaluating ⟨key⟩ is different from every datum, then if there is an else clause its expressions are evaluated and the result(s) of the last is(are) the result(s) of the `case` expression; otherwise the result of the `case` expression is the unspecified value.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⟹  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                 ⟹  the unspecified value
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant))        ⟹  consonant
```

A sample definition of `case` in terms of simpler forms is in appendix B.

`(and` ⟨test$_1$⟩ `...)` syntax

Syntax: The ⟨test⟩s must be expressions. *Semantics:* The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value (see section 9.11) is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then `#t` is returned.

```
(and (= 2 2) (> 2 1))       ⟹  #t
(and (= 2 2) (< 2 1))       ⟹  #f
(and 1 2 'c '(f g))         ⟹  (f g)
(and)                       ⟹  #t
```

The `and` keyword could be defined in terms of `if` using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))
```

`(or` ⟨test$_1$⟩ `...)` syntax

Syntax: The ⟨test⟩s must be expressions. *Semantics:* The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see section 9.11) is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then `#f` is returned.

```
(or (= 2 2) (> 2 1))        ⟹  #t
(or (= 2 2) (< 2 1))        ⟹  #t
(or #f #f #f)               ⟹  #f
(or '(b c) (/ 3 0))         ⟹  (b c)
```

The `or` keyword could be defined in terms of `if` using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))
```

9.5.6. Binding constructs

The four binding constructs `let`, `let*`, `letrec*`, and `letrec` give Scheme a block structure, like Algol 60. The syntax of the four constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially; while in a `letrec*` and in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

In addition, the binding constructs `let-values` and `let*-values` allow the binding of results of expression returning multiple values. They are analogous to `let` and `let*` in the way they establish regions: in a `let-values` expression, the initial values are computed before any of the variables become bound; in a `let*-values` expression, the bindings are performed sequentially.

Note: These forms are compatible with SRFI 11 [25].

`(let` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 9.4. It is a syntax violation for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨init⟩s are evaluated in the current environment (in some unspecified order), the ⟨variable⟩s are bound to fresh locations holding the results, the ⟨body⟩ is evaluated in the extended environment, and the value(s) of the last expression of ⟨body⟩ is(are) returned. Each binding of a ⟨variable⟩ has ⟨body⟩ as its region.

```
(let ((x 2) (y 3))
  (* x y))                  ⟹  6
```

```
(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))                ⟹  35
```

See also named `let`, section 9.19.

`(let*` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

and ⟨body⟩ must be a sequence of one or more expressions.

Semantics: The `let*` form is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (⟨variable⟩ ⟨init⟩) is that part of the `let*` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                ⟹  70
```

The `let*` keyword could be defined in terms of `let` using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 expr1) (name2 expr2) ...)
       body1 body2 ...)
     (let ((name1 expr1))
       (let* ((name2 expr2) ...)
         body1 body2 ...)))))
```

`(letrec` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

and ⟨body⟩ must be a sequence of one or more expressions. It is a syntax violation for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨variable⟩s are bound to fresh locations holding undefined values, the ⟨init⟩s are evaluated in the resulting environment (in some unspecified order), each ⟨variable⟩ is assigned to the result of the corresponding ⟨init⟩, the ⟨body⟩ is evaluated in the resulting environment, and the value(s) of the last expression in ⟨body⟩ is(are) returned. Each binding of a ⟨variable⟩ has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                                 ⟹  #t
```

One restriction on `letrec` is very important: it must be possible to evaluate each ⟨init⟩ without assigning or referring to the value of any ⟨variable⟩. If this restriction is violated, an exception with condition type `&contract` is raised. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the ⟨init⟩s are `lambda` expressions and the restriction is satisfied automatically.

A sample definition of `letrec` in terms of simpler forms is in appendix B.

`(letrec*` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

and ⟨body⟩ must be a sequence of one or more expressions. It is a syntax violation for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨variable⟩s are bound to fresh locations holding undefined values, each ⟨variable⟩ is assigned in left-to-right order to the result of evaluating the corresponding ⟨init⟩, the ⟨body⟩ is evaluated in the resulting environment, and the value(s) of the last expression in ⟨body⟩ is(are) returned. Despite the left-to-right evaluation and assignment order, each binding of a ⟨variable⟩ has the entire `letrec*` expression as its region, making it possible to define mutually recursive procedures.

```
(letrec* ((p
           (lambda (x)
             (+ 1 (q (- x 1)))))
          (q
           (lambda (y)
             (if (zero? y)
                 0
                 (+ 1 (p (- y 1))))))
          (x (p 5))
          (y x))
  y)
                                 ⟹  5
```

One restriction on `letrec*` is very important: it must be possible to evaluate each ⟨init⟩ without assigning or referring to the value the corresponding ⟨variable⟩ or the

⟨variable⟩ of any of the bindings that follow it in ⟨bindings⟩. If this restriction is violated, an exception with condition type `&contract` is raised. The restriction is necessary because Scheme passes arguments by value rather than by name.

The `letrec*` keyword could be defined approximately in terms of `let` and `set!` using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax letrec*
  (syntax-rules ()
    ((letrec* ((var1 init1) ...) body1 body2 ...)
     (let ((var1 <undefined>) ...)
       (set! var1 init1)
       ...
       (let () body1 body2 ...)))))
```

The syntax `<undefined>` represents an expression that returns something that, when stored in a location, causes an exception with condition type `&contract` to be raised if an attempt to read from or write to the location occurs before the assignments generated by the `letrec*` transformation take place. (No such expression is defined in Scheme.)

`(let-values` ⟨mv-bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Mv-bindings⟩ must have the form

`((`⟨formals$_1$⟩ ⟨init$_1$⟩`) ...)`,

and ⟨body⟩ is as described in section 9.4. It is a syntax violation for a variable to appear more than once in the list of variables that appear as part of the formals.

Semantics: The ⟨init⟩s are evaluated in the current environment (in some unspecified order), and the variables occurring in the ⟨formals⟩ are bound to fresh locations containing the values returned by the ⟨init⟩s, where the ⟨formals⟩ are matched to the return values in the same way that the ⟨formals⟩ in a `lambda` expression are matched to the actual arguments in a procedure call. Then, the ⟨body⟩ is evaluated in the extended environment, and the value(s) of the last expression of ⟨body⟩ is(are) returned. Each binding of a variable has ⟨body⟩ as its region. If no ⟨formals⟩ match, an exception with condition type `&contract` is raised.

```
(let-values (((a b) (values 1 2))
             ((c d) (values 3 4)))
  (list a b c d))           ⟹ (1 2 3 4)

(let-values (((a b . c) (values 1 2 3 4)))
  (list a b c))             ⟹ (1 2 (3 4))

(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let-values (((a b) (values x y))
               ((x y) (values a b)))
    (list a b x y)))        ⟹ (x y a b)
```

A sample definition of `let-values` in terms of simpler forms is in appendix B.

`(let*-values` ⟨mv-bindings⟩ ⟨body⟩`)` syntax

The `let*-values` form is the same as with `let-values`, but the bindings are processed sequentially from left to right, and the region of the bindings indicated by `(`⟨formals⟩ ⟨init⟩`)` is that part of the `let*-values` expression to the right of the bindings. Thus, the second set of bindings is evaluated in an environment in which the first set of bindings is visible, and so on.

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y)))        ⟹ (x y x y)
```

The following macro defines `let*-values` in terms of `let` and `let-values`:

```
(define-syntax let*-values
  (syntax-rules ()
    ((let*-values () body1 body2 ...)
     (let () body1 body2 ...))
    ((let*-values (binding1 binding2 ...)
       body1 body2 ...)
     (let-values (binding1)
       (let*-values (binding2 ...)
         body1 body2 ...)))))
```

9.5.7. Sequencing

`(begin` ⟨form⟩ ...`)` syntax
`(begin` ⟨expression⟩ ⟨expression⟩ ...`)` syntax

The ⟨begin⟩ keyword has two different roles, depending on its context:

- It may appear as a form in a ⟨body⟩ (see section 9.4), ⟨library body⟩ (see section 6.1), or ⟨script body⟩ (see chapter 7), or directly nested in a `begin` form that appears in a body. In this case, the `begin` form must have the shape specified in the first header line. This use of `begin` acts as a *splicing* form—the forms inside the ⟨body⟩ are spliced intop the surrounding body, as if the `begin` wrapper were not actually present.

  A `begin` form in a ⟨body⟩ or ⟨library body⟩ must be non-empty if it appears after the first ⟨expression⟩ within the body.

- It may appear as a regular expression and must have the shape specified in the second header line. In this case, the ⟨expression⟩s are evaluated sequentially from left to right, and the value(s) of the last ⟨expression⟩ is(are) returned. This expression type is used to sequence side effects such as assignments or input and output.

```
(define x 0)

(begin (set! x 5)
       (+ x 1))              ⟹  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))    ⟹  unspecified
           and prints  4 plus 1 equals 5
```

The following macro, which uses `syntax-rules` (see section 9.21), defines `begin` in terms of `lambda`. Note that it only covers the expression case of `begin`.

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp ...)
     ((lambda () exp ...)))))
```

The following alternative expansion for `begin` does not make use of the ability to write more than one expression in the body of a lambda expression. It, too, only covers the expression case of `begin`.

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp)
     exp)
    ((begin exp1 exp2 ...)
     (call-with-values
         (lambda () exp1)
       (lambda ignored
         (begin exp2 ...))))))
```

9.6. Equivalence predicates

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, and `equal?` is the coarsest. The `eqv?` predicate is slightly less discriminating than `eq?`.

`(eqv?` *obj*$_1$ *obj*$_2$`)` procedure

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if *obj*$_1$ and *obj*$_2$ should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of `eqv?` holds for all implementations of Scheme.

The `eqv?` procedure returns `#t` if one of the following holds:

- *Obj*$_1$ and *obj*$_2$ are both `#t` or both `#f`.
- *Obj*$_1$ and *obj*$_2$ are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                              ⟹  #t
```

- *Obj*$_1$ and *obj*$_2$ are both exact numbers, and are numerically equal (see `=`, section see section 9.10).
- *Obj*$_1$ and *obj*$_2$ are both inexact numbers, are numerically equal (see `=`, section see section 9.10, and yield the same results (in the sense of `eqv?`) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures.
- *Obj*$_1$ and *obj*$_2$ are both characters and are the same character according to the `char=?` procedure (section 9.14).
- both *Obj*$_1$ and *obj*$_2$ are the empty list.
- *Obj*$_1$ and *obj*$_2$ are pairs, vectors, or strings that denote the same locations in the store (section 4.6).
- *Obj*$_1$ and *obj*$_2$ are procedures whose location tags are equal (section 9.5.2).

The `eqv?` procedure returns `#f` if one of the following holds:

- *Obj*$_1$ and *obj*$_2$ are of different types (section 9.1).
- One of *obj*$_1$ and *obj*$_2$ is `#t` but the other is `#f`.
- *Obj*$_1$ and *obj*$_2$ are symbols but

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                              ⟹  #f
```

- One of *obj*$_1$ and *obj*$_2$ is an exact number but the other is an inexact number.
- *Obj*$_1$ and *obj*$_2$ are rational numbers for which the `=` procedure returns `#f`.
- *Obj*$_1$ and *obj*$_2$ yield different results (in the sense of `eqv?`) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures.
- *Obj*$_1$ and *obj*$_2$ are characters for which the `char=?` procedure returns `#f`.
- One of *obj*$_1$ and *obj*$_2$ is the empty list but the other is not.
- *Obj*$_1$ and *obj*$_2$ are pairs, vectors, or strings that denote distinct locations.

- Obj_1 and obj_2 are procedures that would behave differently (return different value(s) or have different side effects) for some arguments.

```
(eqv? 'a 'a)                    ⟹  #t
(eqv? 'a 'b)                    ⟹  #f
(eqv? 2 2)                      ⟹  #t
(eqv? '() '())                  ⟹  #t
(eqv? 100000000 100000000)      ⟹  #t
(eqv? (cons 1 2) (cons 1 2))    ⟹  #f
(eqv? (lambda () 1)
      (lambda () 2))            ⟹  #f
(eqv? #f 'nil)                  ⟹  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                   ⟹  #t
```

The following examples illustrate cases in which the above rules do not fully specify the behavior of `eqv?`. All that can be said about such cases is that the value returned by `eqv?` must be a boolean.

```
(eqv? "" "")                    ⟹  unspecified
(eqv? '#() '#())                ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))           ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))           ⟹  unspecified
```

The next set of examples shows the use of `eqv?` with procedures that have local state. Calls to `gen-counter` must return a distinct procedure every time, since each procedure has its own internal counter. The `gen-loser` procedure, however, returns equivalent procedures each time, since the local state does not affect the value or side effects of the procedures.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                   ⟹  #t
(eqv? (gen-counter) (gen-counter))
                                ⟹  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                   ⟹  #t
(eqv? (gen-loser) (gen-loser))
                                ⟹  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                                ⟹  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
                                ⟹  #f
```

Since it is the effect of trying to modify constant objects (those returned by literal expressions) is unspecified, implementations are permitted, though not required, to share structure between constants where appropriate. Thus the value of `eqv?` on constants is sometimes implementation-dependent.

```
(eqv? '(a) '(a))                ⟹  unspecified
(eqv? "a" "a")                  ⟹  unspecified
(eqv? '(b) (cdr '(a b)))        ⟹  unspecified
(let ((x '(a)))
  (eqv? x x))                   ⟹  #t
```

Rationale: The above definition of `eqv?` allows implementations latitude in their treatment of procedures and literals: implementations are free either to detect or to fail to detect that two procedures or two literals are equivalent to each other, and can decide whether or not to merge representations of equivalent objects by using the same pointer or bit pattern to represent both.

`(eq?` obj_1 obj_2`)` procedure

The `eq?` predicate is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`.

The `eq?` and `eqv?` predicates are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, and non-empty strings and vectors. The behavior of `eq?` on numbers and characters is implementation-dependent, but it always returns either true or false, and returns true only when `eqv?` would also return true. The `eq?` predicate may also behave differently from `eqv?` on empty vectors and empty strings.

```
(eq? 'a 'a)                     ⟹  #t
(eq? '(a) '(a))                 ⟹  unspecified
(eq? (list 'a) (list 'a))       ⟹  #f
(eq? "a" "a")                   ⟹  unspecified
(eq? "" "")                     ⟹  unspecified
(eq? '() '())                   ⟹  #t
(eq? 2 2)                       ⟹  unspecified
(eq? #\A #\A)                   ⟹  unspecified
(eq? car car)                   ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                    ⟹  unspecified
(let ((x '(a)))
  (eq? x x))                    ⟹  #t
(let ((x '#()))
  (eq? x x))                    ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                    ⟹  #t
```

Rationale: It is usually be possible to implement `eq?` much more efficiently than `eqv?`, for example, as a simple pointer comparison instead of as some more complicated operation. One reason is that it may not be possible to compute `eqv?` of two

numbers in constant time, whereas `eq?` implemented as pointer comparison will always finish in constant time. The `eq?` predicate may be used like `eqv?` in applications using procedures to implement objects with state since it obeys the same constraints as `eqv?`.

`(equal?` *obj*$_1$ *obj*$_2$`)` procedure

The `equal?` predicate returns `#t` if and only if the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees.

The `equal?` predicate treats pairs and vectors as nodes with outgoing edges, uses `string=?` to compare strings, uses `bytes=?` to compare bytes objects (see chapter 11), and uses `eqv?` to compare other nodes.

```
(equal? 'a 'a)              ⟹  #t
(equal? '(a) '(a))          ⟹  #t
(equal? '(a (b) c)
        '(a (b) c))         ⟹  #t
(equal? "abc" "abc")        ⟹  #t
(equal? 2 2)                ⟹  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a)) ⟹  #t
(equal? '#vu8(1 2 3 4 5)
        (u8-list->bytes
         '(1 2 3 4 5))      ⟹  #t
(equal? (lambda (x) x)
        (lambda (y) y))     ⟹  unspecified

(let* ((x (list 'a))
       (y (list 'a))
       (z (list x y)))
  (list (equal? z (list y x))
        (equal? z (list x x))))
          ⟹  (#t #t)
```

9.7. Procedure predicate

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

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

```
(procedure? car)            ⟹  #t
(procedure? 'car)           ⟹  #f
(procedure? (lambda (x) (* x x)))
                            ⟹  #t
(procedure? '(lambda (x) (* x x)))
                            ⟹  #f
```

9.8. Unspecified value

`(unspecified)` procedure

Returns the unspecified value. (See section 9.1.)

Note: The unspecified value is not a datum value, and thus has no external representation.

`(unspecified?` *obj*`)` procedure

Returns `#t` if *obj* is the unspecified value, otherwise returns `#f`.

9.9. End of file object

The end of file object is returned by various I/O procedures (see section 15.3) when they reach end of file.

`(eof-object)` procedure

Returns the end of file object.

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

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

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

9.10. Generic arithmetic

The procedures described here implement arithmetic that is generic over the numerical tower described in chapter 2. Unlike the procedures exported by the libraries described in chapter 16, the generic procedures described in this section accept both exact and inexact numbers as arguments, performing coercions and selecting the appropriate operations as determined by the numeric subtypes of their arguments.

Chapter 2 contains a detailed description of the numerical types of Scheme and a discussion of the concept of exactness. Chapter 16 contains the mathematical definitions of some operations that are assumed here, and describes libraries that define other numerical procedures.

9.10.1. Propagation of exactness and inexactness

The procedures listed below must return the correct exact result provided all their arguments are exact:

```
+           -           *
max         min         abs
numerator   denominator gcd
lcm         floor       ceiling
truncate    round       rationalize
expt        real-part   imag-part
make-rectangular
```

The procedures listed below must return the correct exact result provided all their arguments are exact, and no divisors are zero:

```
/
div          mod          div+mod
div0         mod0         div0+mod0
```

The general rule is that the generic operations return the correct exact result when all of their arguments are exact and the result is mathematically well-defined, but return an inexact result when any argument is inexact. Exceptions to this rule include `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `expt`, `make-polar`, `magnitude`, and `angle`, which are allowed (but not required) to return inexact results even when given exact arguments, as indicated in the specification of these procedures.

One general exception to the rule above is that an implementation may return an exact result despite inexact arguments if that exact result would be the correct result for all possible substitutions of exact arguments for the inexact ones.

9.10.2. Numerical operations

On exact arguments, the procedures described here behave consistently with the corresponding procedures of section 16.5 whose names are prefixed by `exact-` or `exact`. On inexact arguments, the procedures described here behave consistently with the corresponding procedures of section 16.6 whose names are prefixed by `inexact-` or `inexact`.

Numerical type predicates

`(number?` *obj*`)` procedure
`(complex?` *obj*`)` procedure
`(real?` *obj*`)` procedure
`(rational?` *obj*`)` procedure
`(integer?` *obj*`)` procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is of the named type, and otherwise they return `#f`. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

If z is a complex number, then `(real?` z`)` is true if and only if `(zero? (imag-part` z`))` and `(exact? (imag-part` z`))` are both true.

If x is a real number, then `(rational?` x`)` is true if and only if there exist exact integers k_1 and k_2 such that `(=` x `(/` k_1 k_2`))` and `(= (numerator` x`)` k_1`)` and `(= (denominator` x`)` k_2`)` are all true. Thus infinities and NaNs are not rational numbers.

If q is a rational number, then `(integer?` q`)` is true if and only if `(= (denominator` q`) 1)` is true. If q is not a rational number, then `(integer?` q`)` is false.

```
(complex? 3+4i)            ⟹  #t
(complex? 3)               ⟹  #t
(real? 3)                  ⟹  #t
(real? -2.5+0.0i)          ⟹  #f
(real? -2.5+0i)            ⟹  #t
(real? -2.5)               ⟹  #t
(real? #e1e10)             ⟹  #t
(rational? 6/10)           ⟹  #t
(rational? 6/3)            ⟹  #t
(rational? 2)              ⟹  #t
(integer? 3+0i)            ⟹  #t
(integer? 3.0)             ⟹  #t
(integer? 8/4)             ⟹  #t

(number? +nan.0)           ⟹  #t
(complex? +nan.0)          ⟹  #t
(real? +nan.0)             ⟹  #t
(rational? +nan.0)         ⟹  #f
(complex? +inf.0)          ⟹  #t
(real? -inf.0)             ⟹  #t
(rational? -inf.0)         ⟹  #f
(integer? -inf.0)          ⟹  #f
```

Note: The behavior of these type predicates on inexact numbers is unreliable, because any inaccuracy may affect the result.

`(real-valued?` *obj*`)` procedure
`(rational-valued?` *obj*`)` procedure
`(integer-valued?` *obj*`)` procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is a number and is equal in the sense of `=` to some object of the named type, and otherwise they return `#f`.

```
(real-valued? +nan.0)      ⟹  #f
(real-valued? -inf.0)      ⟹  #t
(real-valued? 3)           ⟹  #t
(real-valued? -2.5+0.0i)   ⟹  #t
(real-valued? -2.5+0i)     ⟹  #t
(real-valued? -2.5)        ⟹  #t
(real-valued? #e1e10)      ⟹  #t

(rational-valued? +nan.0)  ⟹  #f
(rational-valued? -inf.0)  ⟹  #f
(rational-valued? 6/10)    ⟹  #t
(rational-valued? 6/10+0.0i) ⟹ #t
(rational-valued? 6/10+0i) ⟹  #t
(rational-valued? 6/3)     ⟹  #t

(integer-valued? 3+0i)     ⟹  #t
(integer-valued? 3+0.0i)   ⟹  #t
(integer-valued? 3.0)      ⟹  #t
(integer-valued? 3.0+0.0i) ⟹  #t
(integer-valued? 8/4)      ⟹  #t
```

Note: The behavior of these type predicates on inexact numbers is unreliable, because any inaccuracy may affect the result.

`(exact?` z`)` procedure
`(inexact?` z`)` procedure

These numerical predicates provide tests for the exactness of a quantity. For any Scheme number, precisely one of these predicates is true.

```
(exact? 5)              ⟹  #t
(inexact? +inf.0)       ⟹  #t
```

Generic conversions

`(->inexact` z`)` procedure
`(->exact` z`)` procedure

`->inexact` returns an inexact representation of z. If inexact numbers of the appropriate type have bounded precision, then the value returned is an inexact number that is nearest to the argument. If an exact argument has no reasonably close inexact equivalent, an exception with condition type `&implementation-violation` may be raised.

`->exact` returns an exact representation of z. The value returned is the exact number that is numerically closest to the argument; in most cases, the result of this procedure should be numerically equal to its argument. If an inexact argument has no reasonably close exact equivalent, an exception with condition type `&implementation-violation` may be raised.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

`->inexact` and `->exact` are idempotent.

`(real->flonum` x`)` procedure

Returns a flonum representation of x.

The value returned is a flonum that is numerically closest to the argument.

Rationale: The flonums are a subset of the inexact reals, but may be a proper subset. The `real->flonum` procedure converts an arbitrary real to the flonum type required by flonum-specific procedures.

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

`(real->single` x`)` procedure
`(real->double` x`)` procedure

Given a real number x, these procedures compute the best IEEE-754 single or double precision approximation to x and return that approximation as an inexact real.

Note: Both of the two conversions performed by these procedures (to IEEE-754 single or double, and then to an inexact real) may lose precision, introduce error, or may underflow or overflow.

Rationale: The ability to round to IEEE-754 single or double precision is occasionally needed for control of precision or for interoperability.

Arithmetic operations

`(=` z_1 z_2 z_3 ...`)` procedure
`(<` $x1$ x_2 x_3 ...`)` procedure
`(>` $x1$ x_2 x_3 ...`)` procedure
`(<=` $x1$ x_2 x_3 ...`)` procedure
`(>=` $x1$ x_2 x_3 ...`)` procedure

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

```
(= +inf.0 +inf.0)       ⟹  #t
(= -inf.0 +inf.0)       ⟹  #f
(= -inf.0 -inf.0)       ⟹  #t
```

For any real number x that is neither infinite nor NaN:

```
(< -inf.0 x +inf.0))    ⟹  #t
(> +inf.0 x -inf.0))    ⟹  #t
```

These predicates are required to be transitive.

Note: The traditional implementations of these predicates in Lisp-like languages are not transitive.

Note: While it is possible to compare inexact numbers using these predicates, the results may be unreliable because a small inaccuracy may affect the result; this is especially true of `=` and `zero?`.

When in doubt, consult a numerical analyst.

`(zero?` z`)` procedure
`(positive?` x`)` procedure
`(negative?` x`)` procedure
`(odd?` n`)` procedure
`(even?` n`)` procedure
`(finite?` x`)` procedure
`(infinite?` x`)` procedure
`(nan?` x`)` procedure

These numerical predicates test a number for a particular property, returning `#t` or `#f`. See note above. The

`zero?` procedure tests if the number is `=` to zero, `positive?` tests if it is greater than zero, `negative?` tests if it is less than zero, `odd?` tests if it is odd, `even?` tests if it is even, `finite?` tests if it is not an infinity and not a NaN, `infinite?` tests if it is an infinity, `nan?` tests if it is a NaN.

```
(positive? +inf.0)          ⟹  #t
(negative? -inf.0)          ⟹  #t
(finite? +inf.0)            ⟹  #f
(finite? 5)                 ⟹  #t
(finite? 5.0)               ⟹  #t
(infinite? 5.0)             ⟹  #f
(infinite? +inf.0)          ⟹  #t
```

`(max` x_1 x_2 ...`)` procedure
`(min` x_1 x_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)                   ⟹  4    ; exact
(max 3.9 4)                 ⟹  4.0  ; inexact
```

For any real number x:

```
(max +inf.0 x)              ⟹  +inf.0
(min -inf.0 x)              ⟹  -inf.0
```

Note: If any argument is inexact, then the result is also inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If `min` or `max` is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may raise an exception with condition type `&implementation-restriction`.

`(+` z_1 ...`)` procedure
`(*` z_1 ...`)` procedure

These procedures return the sum or product of their arguments.

```
(+ 3 4)                     ⟹  7
(+ 3)                       ⟹  3
(+)                         ⟹  0
(+ +inf.0 +inf.0)           ⟹  +inf.0
(+ +inf.0 -inf.0)           ⟹  +nan.0

(* 4)                       ⟹  4
(*)                         ⟹  1
(* 5 +inf.0)               ⟹  +inf.0
(* -5 +inf.0)              ⟹  -inf.0
(* +inf.0 +inf.0)          ⟹  +inf.0
(* +inf.0 -inf.0)          ⟹  -inf.0
(* 0 +inf.0)               ⟹  0 or +nan.0
(* 0 +nan.0)               ⟹  0 or +nan.0
```

For any real number x that is neither infinite nor NaN:

```
(+ +inf.0 x)                ⟹  +inf.0
(+ -inf.0 x)                ⟹  -inf.0
(+ +nan.0 x)                ⟹  +nan.0
```

For any real number x that is neither infinite nor NaN nor an exact 0:

```
(* +nan.0 x)                ⟹  +nan.0
```

If any of these procedures are applied to mixed non-rational real and non-real complex arguments, they either raise an exception with condition type `&implementation-restriction` or return an unspecified number.

`(-` z`)` procedure
`(-` z_1 z_2 ...`)` procedure
`(/` z`)` procedure
`(/` z_1 z_2 ...`)` procedure

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

```
(- 3 4)                     ⟹  -1
(- 3 4 5)                   ⟹  -6
(- 3)                       ⟹  -3
(- +inf.0 +inf.0)           ⟹  +nan.0

(/ 3 4 5)                   ⟹  3/20
(/ 3)                       ⟹  1/3
(/ 0.0)                     ⟹  +inf.0
(/ 1.0 0)                   ⟹  +inf.0
(/ -1 0.0)                  ⟹  -inf.0
(/ +inf.0)                  ⟹  0.0
(/ 0 0)
          ⟹ &contract exception or +nan.0
(/ 0 3.5)                   ⟹  0.0 ; inexact
(/ 0 0.0)                   ⟹  +nan.0
(/ 0.0 0)                   ⟹  +nan.0
(/ 0.0 0.0)                 ⟹  +nan.0
```

If any of these procedures are applied to mixed non-rational real and non-real complex arguments, they either raise an exception with condition type `&implementation-restriction` or return an unspecified number.

`(abs` x`)` procedure

Returns the absolute value of its argument.

```
(abs -7)                    ⟹  7
(abs -inf.0)                ⟹  +inf.0
```

`(div+mod` x_1 x_2`)` procedure
`(div` x_1 x_2`)` procedure
`(mod` x_1 x_2`)` procedure

`(div0+mod0` x_1 x_2`)` procedure
`(div0` x_1 x_2`)` procedure
`(mod0` x_1 x_2`)` procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in section 16.2.1. In each case, x_1 must be neither infinite nor a NaN, and x_2 must be nonzero; otherwise, an exception with condition type `&contract` is raised.

```
(div x1 x2)                  ⟹ x1 div x2
(mod x1 x2)                  ⟹ x1 mod x2
(div+mod x1 x2)              ⟹ x1 div x2, x1 mod x2
                                ; two return values
(div0 x1 x2)                 ⟹ x1 div0 x2
(mod0 x1 x2)                 ⟹ x1 mod0 x2
(div0+mod0 x1 x2)
          ⟹ x1 div0 x2, x1 mod0 x2
                             ; two return values
```

`(gcd` n_1 ...`)` procedure
`(lcm` n_1 ...`)` procedure

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative.

```
(gcd 32 -36)                 ⟹ 4
(gcd)                        ⟹ 0
(lcm 32 -36)                 ⟹ 288
(lcm 32.0 -36)               ⟹ 288.0 ; inexact
(lcm)                        ⟹ 1
```

`(numerator` q`)` procedure
`(denominator` q`)` procedure

These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

```
(numerator (/ 6 4))          ⟹ 3
(denominator (/ 6 4))        ⟹ 2
(denominator
 (->inexact (/ 6 4)))        ⟹ 2.0
```

`(floor` x`)` procedure
`(ceiling` x`)` procedure
`(truncate` x`)` procedure
`(round` x`)` procedure

These procedures return inexact integers on inexact arguments that are not infinities or NaNs, and exact integers on exact rational arguments. For such arguments, `floor` returns the largest integer not larger than x. The `ceiling` procedure returns the smallest integer not smaller than x. The `truncate` procedure returns the integer closest to x whose absolute value is not larger than the absolute value of x. The `round` procedure returns the closest integer to x, rounding to even when x is halfway between two integers.

Rationale: The `round` procedure rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

Note: If the argument to one of these procedures is inexact, then the result is also inexact. If an exact value is needed, the result should be passed to the `->exact` procedure.

Although infinities and NaNs are not integers, these procedures return an infinity when given an infinity as an argument, and a NaN when given a NaN.

```
(floor -4.3)                 ⟹ -5.0
(ceiling -4.3)               ⟹ -4.0
(truncate -4.3)              ⟹ -4.0
(round -4.3)                 ⟹ -4.0

(floor 3.5)                  ⟹ 3.0
(ceiling 3.5)                ⟹ 4.0
(truncate 3.5)               ⟹ 3.0
(round 3.5)                  ⟹ 4.0  ; inexact

(round 7/2)                  ⟹ 4    ; exact
(round 7)                    ⟹ 7

(floor +inf.0)               ⟹ +inf.0
(ceiling -inf.0)             ⟹ -inf.0
(round +nan.0)               ⟹ +nan.0
```

`(rationalize` x_1 x_2`)` procedure

The `rationalize` procedure returns the *simplest* rational number differing from x_1 by no more than x_2. A rational number r_1 is *simpler* than another rational number r_2 if $r_1 = p_1/q_1$ and $r_2 = p_2/q_2$ (in lowest terms) and $|p_1| \leq |p_2|$ and $|q_1| \leq |q_2|$. Thus $3/5$ is simpler than $4/7$. Although not all rationals are comparable in this ordering (consider $2/7$ and $3/5$) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler $2/5$ lies between $2/7$ and $3/5$). Note that $0 = 0/1$ is the simplest rational of all.

```
(rationalize
  (->exact .3) 1/10)         ⟹ 1/3    ; exact
(rationalize .3 1/10)        ⟹ #i1/3  ; inexact

(rationalize +inf.0 3)       ⟹ +inf.0
(rationalize +inf.0 +inf.0)  ⟹ +nan.0
(rationalize 3 +inf.0)       ⟹ 0.0
```

`(exp` z`)` procedure
`(log` z`)` procedure
`(log` z_1 z_2`)` procedure
`(sin` z`)` procedure

`(cos` z`)` procedure
`(tan` z`)` procedure
`(asin` z`)` procedure
`(acos` z`)` procedure
`(atan` z`)` procedure
`(atan` x_1 x_2`)` procedure

These procedures compute the usual transcendental functions. The `exp` procedure computes the base-e exponential of z. The `log` procedure with a single argument computes the natural logarithm of z (not the base ten logarithm); `(log` z_1 z_2`)` computes the base-z_2 logarithm of z_1. The `asin`, `acos`, and `atan` procedures compute arcsine, arccosine, and arctangent, respectively. The two-argument variant of `atan` computes `(angle (make-rectangular` x_2 x_1`))`.

See section 16.2.2 for the underlying mathematical operations. These procedures may return inexact results even when given exact arguments.

```
(exp +inf.0)                ⟹ +inf.0
(exp -inf.0)                ⟹ 0.0
(log +inf.0)                ⟹ +inf.0
(log 0.0)                   ⟹ -inf.0
(log 0)
          ⟹ &contract exception
(log -inf.0)                ⟹ +inf.0+πi
(atan -inf.0)
          ⟹ -1.5707963267948965 ; approximately
(atan +inf.0)
          ⟹ 1.5707963267948965 ; approximately
(log -1.0+0.0i)             ⟹ 0.0+πi
(log -1.0-0.0i)             ⟹ 0.0-πi
          ; if -0.0 is distinguished
```

`(sqrt` z`)` procedure

Returns the principal square root of z. For rational z, the result has either positive real part, or zero real part and non-negative imaginary part. With log defined as in section 16.2.2, the value of `(sqrt` z`)` could be expressed as

$$e^{\frac{\log z}{2}}.$$

The `sqrt` procedure may return an inexact result even when given an exact argument.

```
(sqrt -5)
          ⟹  0.0+2.236067977749979i ; approximately
(sqrt +inf.0)               ⟹  +inf.0
(sqrt -inf.0)               ⟹  +inf.0i
```

`(expt` z_1 z_2`)` procedure

Returns z_1 raised to the power z_2. For nonzero z_1,

$$z_1{}^{z_2} = e^{z_2 \log z_1}$$

0.0^z is 1.0 if $z = 0.0$, and 0.0 if `(real-part` z`)` is positive. For other cases in which the first argument is zero, an exception is raised with condition type `&implementation-restriction` or an unspecified number is returned.

For an exact z_1 and an exact integer z_2, `(expt` z_1 z_2`)` must return an exact result. For all other values of z_1 and z_2, `(expt` z_1 z_2`)` may return an inexact result, even when both z_1 and z_2 are exact.

```
(expt 5 3)                  ⟹  125
(expt 5 -3)                 ⟹  1/125
(expt 5 0)                  ⟹  1
(expt 0 5)                  ⟹  0
(expt 0 5+.0000312i)        ⟹  0
(expt 0 -5)                 ⟹  unspecified
(expt 0 -5+.0000312i)       ⟹  unspecified
(expt 0 0)                  ⟹  1
(expt 0.0 0.0)              ⟹  1.0
```

`(make-rectangular` x_1 x_2`)` procedure
`(make-polar` x_3 x_4`)` procedure
`(real-part` z`)` procedure
`(imag-part` z`)` procedure
`(magnitude` z`)` procedure
`(angle` z`)` procedure

Suppose x_1, x_2, x_3, and x_4 are real numbers and z is a complex number such that

$$z = x_1 + x_2 i = x_3 e^{i x_4}.$$

Then:

```
(make-rectangular x1 x2)    ⟹  z
(make-rectangular x3 x4)    ⟹  z
(real-part z)               ⟹  x1
(imag-part z)               ⟹  x2
(magnitude z)               ⟹  |x3|
(angle z)                   ⟹  xangle
```

where $-\pi \leq x_{\text{angle}} \leq \pi$ with $x_{\text{angle}} = x_4 + 2\pi n$ for some integer n.

```
(angle -1.0)                ⟹  π
(angle -1.0+0.0)            ⟹  π
(angle -1.0-0.0)            ⟹  -π
          ; if -0.0 is distinguished
```

Moreover, suppose x_1, x_2 are such that either x_1 or x_2 is an infinity, then

```
(make-rectangular x1 x2)    ⟹  z
(magnitude z)               ⟹  +inf.0
```

The `make-polar`, `magnitude`, and `angle` procedures may return inexact results even when given exact arguments.

```
(angle -1)                  ⟹  π
(angle +inf.0)              ⟹  0.0
(angle -inf.0)              ⟹  π
(angle -1.0+0.0)            ⟹  π
(angle -1.0-0.0)            ⟹  −π
          ; if -0.0 is distinguished
```

Numerical Input and Output

`(number->string` *z*`)` procedure
`(number->string` *z radix*`)` procedure
`(number->string` *z radix precision*`)` procedure

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. If a *precision* is specified, then *z* must be an inexact complex number, *precision* must be an exact positive integer, and *radix* must be 10. The `number->string` procedure takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number
                                        radix)
                        radix)))
```

is true. If no possible result makes this expression true, an exception with condition type `&implementation-restriction` is raised.

If a *precision* is specified, then the representations of the inexact real components of the result, unless they are infinite or NaN, specify an explicit ⟨mantissa width⟩ p, and p is the least $p \geq$ *precision* for which the above expression is true.

If *z* is inexact, the radix is 10, and the above expression and condition can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent, trailing zeroes, and mantissa width) needed to make the above expression and condition true [5, 8]; otherwise the format of the result is unspecified.

The result returned by `number->string` never contains an explicit radix prefix.

Note: The error case can occur only when *z* is not a complex number or is a complex number with a non-rational real or imaginary part.

Rationale: If *z* is an inexact number represented using binary floating point, and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and representations other than binary floating point.

`(string->number` *string*`)` procedure
`(string->number` *string radix*`)` procedure

Returns a number of the maximally precise representation expressed by the given *string*. *Radix* must be an exact integer, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g. `"#o177"`). If *radix* is not supplied, then the default radix is 10. If *string* is not a syntactically valid notation for a number, then `string->number` returns `#f`.

```
(string->number "100")      ⟹  100
(string->number "100" 16)   ⟹  256
(string->number "1e2")      ⟹  100.0
(string->number "15##")     ⟹  1500.0
(string->number "+inf.0")   ⟹  +inf.0
(string->number "-inf.0")   ⟹  -inf.0
(string->number "+nan.0")   ⟹  +nan.0
```

9.11. Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. What really matters, though, are the objects that the Scheme conditional expressions (`if`, `cond`, `and`, `or`, `do`) treat as true or false. The phrase "a true value" (or sometimes just "true") means any object treated as true by the conditional expressions, and the phrase "a false value" (or "false") means any object treated as false by the conditional expressions.

Of all the standard Scheme values, only `#f` counts as false in conditional expressions. Except for `#f`, all standard Scheme values, including `#t`, pairs, the empty list, symbols, numbers, strings, vectors, and procedures, count as true.

Note: Programmers accustomed to other dialects of Lisp should be aware that Scheme distinguishes both `#f` and the empty list from the symbol `nil`.

`(not` *obj*`)` procedure

Returns `#t` if *obj* is false, and returns `#f` otherwise.

```
(not #t)              ⟹  #f
(not 3)               ⟹  #f
(not (list 3))        ⟹  #f
(not #f)              ⟹  #t
(not '())             ⟹  #f
(not (list))          ⟹  #f
(not 'nil)            ⟹  #f
```

`(boolean?` *obj*`)` procedure

Returns `#t` if *obj* is either `#t` or `#f` and returns `#f` otherwise.

```
(boolean? #f)         ⟹  #t
(boolean? 0)          ⟹  #f
(boolean? '())        ⟹  #f
```

9.12. Pairs and lists

A *pair* (sometimes called a *dotted pair*) is a record structure with two fields called the car and cdr fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`.

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set X such that

- The empty list is in X.
- If *list* is in X, then any pair whose cdr field contains *list* is also in X.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type (it is not a pair); it has no elements and its length is zero.

Note: The above definitions imply that all lists have finite length and are terminated by the empty list.

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

```
(a b c . d)
```

is equivalent to

```
(a . (b . (c . d)))
```

Whether a given pair is a list depends upon what is stored in the cdr field.

`(pair?` *obj*`)` procedure

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

```
(pair? '(a . b))          ⟹  #t
(pair? '(a b c))          ⟹  #t
(pair? '())               ⟹  #f
(pair? '#(a b))           ⟹  #f
```

`(cons` obj_1 obj_2`)` procedure

Returns a newly allocated pair whose car is obj_1 and whose cdr is obj_2. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

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

`(car` *pair*`)` procedure

Returns the contents of the car field of *pair*.

```
(car '(a b c))            ⟹  a
(car '((a) b c d))        ⟹  (a)
(car '(1 . 2))            ⟹  1
(car '())                 ⟹  &contract exception
```

`(cdr` *pair*`)` procedure

Returns the contents of the cdr field of *pair*.

```
(cdr '((a) b c d))        ⟹  (b c d)
(cdr '(1 . 2))            ⟹  2
(cdr '())                 ⟹  &contract exception
```

`(caar` *pair*`)` procedure
`(cadr` *pair*`)` procedure
⋮ ⋮
`(cdddar` *pair*`)` procedure
`(cddddr` *pair*`)` procedure

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

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

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

`(null?` *obj*`)` procedure

Returns `#t` if *obj* is the empty list, otherwise returns `#f`.

`(list?` *obj*`)` procedure

Returns `#t` if *obj* is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list.

```
(list? '(a b c))          ⟹  #t
(list? '())               ⟹  #t
(list? '(a . b))          ⟹  #f
```

`(list` *obj* ...`)` procedure

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)      ⟹  (a 7 c)
(list)                    ⟹  ()
```

`(length` *list*`)` procedure

Returns the length of *list*.

```
(length '(a b c))           ⟹  3
(length '(a (b) (c d e)))   ⟹  3
(length '())                ⟹  0
```

`(append` *list* ... *obj*`)` procedure

Returns a possibly improper list consisting of the elements of the first *list* followed by the elements of the other *list*s, with *obj* as the cdr of the final pair. An improper list results if *obj* is not a proper list.

```
(append '(x) '(y))          ⟹  (x y)
(append '(a) '(b c d))      ⟹  (a b c d)
(append '(a (b)) '((c)))    ⟹  (a (b) (c))
(append '(a b) '(c . d))    ⟹  (a b c . d)
(append '() 'a)             ⟹  a
```

The resulting improper list is always newly allocated, except that it shares structure with the *obj* argument.

`(reverse` *list*`)` procedure

Returns a newly allocated list consisting of the elements of *list* in reverse order.

```
(reverse '(a b c))          ⟹  (c b a)
(reverse '(a (b c) d (e (f))))
          ⟹  ((e (f)) d (b c) a)
```

`(list-tail` *l k*`)` procedure

If *k* is 0, *l* must be the empty list or a pair. Otherwise, *l* must be a chain of pairs of size at least *k*.

The `list-tail` procedure returns the subchain of pairs of *l* obtained by omitting the first *k* elements.

```
(list-tail '(a b c d) 2)    ⟹  (c d)
(list-tail '(a b c . d) 2)  ⟹  (c . d)
```

`(list-ref` *l k*`)` procedure

L must be a chain of pairs of size at least $k + 1$.

Returns the *k*th element of *l*.

```
(list-ref '(a b c d) 2)     ⟹ c
(list-ref '(a b c . d) 2)   ⟹ c
```

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

The *list*s must all have the same length. *proc* must be a procedure. If the *list*s are non-empty, *proc* must take as many arguments as there are *list*s and must return a single value.

The `map` procedure applies *proc* element-wise to the elements of the *list*s and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *list*s is unspecified.

```
(map cadr '((a b) (d e) (g h)))
          ⟹  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
          ⟹  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))   ⟹  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))               ⟹  (1 2) or (2 1)
```

`(for-each` *proc list*$_1$ *list*$_2$ ...`)` procedure

The *list*s must all have the same length. *proc* must be a procedure. If the *list*s are non-empty, *proc* must take as many arguments as there are *list*s. The `for-each` procedure applies *proc* element-wise to the elements of the *list*s for its side effects, in order from the first element(s) to the last. On the last elements of the *list*s, `for-each` tail-calls *proc*. If the *list*s are empty, `for-each` returns the unspecified value.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                          ⟹  #(0 1 4 9 16)

(for-each (lambda (x) x) '(1 2 3 4))
          ⟹ 4

(for-each even? '())          ⟹ the unspecified value
```

9.13. Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eq?`, `eqv?` and `equal?`) if and only if their names are spelled the same way. This is exactly the property needed to represent identifiers in programs, and so most implementations of Scheme use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated values are used in C and Pascal.

A symbol literal is formed using `quote`.

```
Hello                       ⟹ Hello
'H\x65;llo                  ⟹ Hello
'λ                          ⟹ λ
'\x3BB;                     ⟹ λ
(string->symbol "a b")      ⟹ a\x20;b
(string->symbol "a\\b")     ⟹ a\x5C;b
'a\x20;b                    ⟹ a\x20;b
'|a b|    ; syntax violation
```

```
          ; (illegal character
          ; vertical bar)
'a\nb     ; syntax violation
          ; (illegal use of backslash)
'a\x20    ; syntax violation
          ; (missing semi-colon to
          ; terminate \x escape)
```

`(symbol?` *obj*`)` procedure

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

```
(symbol? 'foo)              ⟹  #t
(symbol? (car '(a b)))      ⟹  #t
(symbol? "bar")             ⟹  #f
(symbol? 'nil)              ⟹  #t
(symbol? '())               ⟹  #f
(symbol? #f)                ⟹  #f
```

`(symbol->string` *symbol*`)` procedure

Returns the name of *symbol* as a string. The returned string may be immutable.

```
(symbol->string 'flying-fish)
                            ⟹  "flying-fish"
(symbol->string 'Martin)    ⟹  "Martin"
(symbol->string
   (string->symbol "Malvina"))
                            ⟹  "Malvina"
```

`(string->symbol` *string*`)` procedure

Returns the symbol whose name is *string*.

```
(eq? 'mISSISSIppi 'mississippi)
          ⟹  #f
(string->symbol "mISSISSIppi")
          ⟹  the symbol with name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
          ⟹  #t
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
          ⟹  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
          ⟹  #t
```

9.14. Characters

Characters are objects that represent Unicode scalar values [51].

Note: Unicode defines a standard mapping between sequences of *code points* (integers in the range 0 to #x10FFFF in the latest version of the standard) and human-readable "characters." More precisely, Unicode distinguishes between glyphs, which are printed for humans to read, and characters, which are abstract entities that map to glyphs (sometimes in a way that's sensitive to surrounding characters). Furthermore, different sequences of code points sometimes correspond to the same character. The relationships among code points, characters, and glyphs are subtle and complex.

Despite this complexity, most things that a literate human would call a "character" can be represented by a single code point in Unicode (though there may exist code-point sequences that represent that same character). For example, Roman letters, Cyrillic letters, Hebrew consonants, and most Chinese characters fall into this category. Thus, the "code point" approximation of "character" works well for many purposes. More specifically, Scheme characters correspond to Unicode *scalar values*, which includes all code points except those designated as surrogates. A *surrogate* is a code point in the range #xD800 to #xDFFF that is used in pairs in the UTF-16 encoding to encode a supplementary character (whose code is in the range #x10000 to #x10FFFF).

`(char?` *obj*`)` procedure

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

`(char->integer` *char*`)` procedure
`(integer->char` *sv*`)` procedure

Sv must be a scalar value, i.e. a non-negative exact integer in $[0, \#x\text{D7FF}] \cup [\#x\text{E000}, \#x\text{10FFFF}]$.

Given a character, `char->integer` returns its scalar value as an exact integer. For a scalar value *sv*, `integer->char` returns its associated character.

```
(integer->char 32)          ⟹ #\space
(char->integer (integer->char 5000))
                            ⟹ 5000
(integer->char #xD800)      ⟹ &contract exception
```

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

These procedures impose a total ordering on the set of characters according to their scalar values.

```
(char<? #\z #\ß)            ⟹ #t
(char<? #\z #\Z)            ⟹ #f
```

9.15. Strings

Strings are sequences of characters.

The *length* of a string is the number of characters that it contains. This number is an exact, non-negative integer

that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

In phrases such as "the characters of *string* beginning with index *start* and ending with index *end*," it is understood that the index *start* is inclusive and the index *end* is exclusive. Thus if *start* and *end* are the same index, a null substring is referred to, and if *start* is zero and *end* is the length of *string*, then the entire string is referred to.

`(string?` *obj*`)` procedure

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

`(make-string` *k*`)` procedure
`(make-string` *k char*`)` procedure

Returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

`(string` *char ...*`)` procedure

Returns a newly allocated string composed of the arguments.

`(string-length` *string*`)` procedure

Returns the number of characters in the given *string*.

`(string-ref` *string k*`)` procedure

K must be a valid index of *string*. The `string-ref` procedure returns character *k* of *string* using zero-origin indexing.

`(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 the unspecified value.

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

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)     ⟹  the unspecified value
(string-set! (g) 0 #\?)     ⟹  unspecified
             ; should raise &contract exception
(string-set! (symbol->string 'immutable)
             0
             #\?)           ⟹  unspecified
             ; should raise &contract exception
```

`(string=?` *string*$_1$ *string*$_2$ *string*$_3$ *...*`)` procedure

Returns `#t` if the strings are the same length and contain the same characters in the same positions, otherwise returns `#f`.

```
(string=? "Strae" "Strasse")⟹ #f
```

`(string<?` *string*$_1$ *string*$_2$ *string*$_3$ *...*`)` procedure
`(string>?` *string*$_1$ *string*$_2$ *string*$_3$ *...*`)` procedure
`(string<=?` *string*$_1$ *string*$_2$ *string*$_3$ *...*`)` procedure
`(string>=?` *string*$_1$ *string*$_2$ *string*$_3$ *...*`)` procedure

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

```
(string<? "z" "ß")          ⟹ #t
(string<? "z" "zz")         ⟹ #t
(string<? "z" "Z")          ⟹ #f
```

`(substring` *string start end*`)` procedure

String must be a string, and *start* and *end* must be exact integers satisfying

$$0 \le start \le end \le \texttt{(string-length}\ string\texttt{)}.$$

The `substring` procedure returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

`(string-append` *string ...*`)` procedure

Returns a newly allocated string whose characters form the concatenation of the given strings.

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

List must be a list of characters. The `string->list` procedure returns a newly allocated list of the characters that make up the given string. The `list->string` procedure returns a newly allocated string formed from the characters in *list*. The `string->list` and `list->string` procedures are inverses so far as `equal?` is concerned.

`(string-copy` *string*`)` procedure

Returns a newly allocated copy of the given *string*.

`(string-fill!` *string char*`)` procedure

Stores *char* in every element of the given *string* and returns the unspecified value.

9.16. Vectors

Vectors are heterogenous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indexes* of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Like list constants, vector constants must be quoted:

```
'#(0 (2 2 2 2) "Anna")
          ⟹  #(0 (2 2 2 2) "Anna")
```

`(vector?` *obj*`)` procedure

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

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

Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

`(vector` *obj* ...`)` procedure

Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)          ⟹  #(a b c)
```

`(vector-length` *vector*`)` procedure

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

`(vector-ref` *vector k*`)` procedure

K must be a valid index of *vector*. The `vector-ref` procedure returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5)
          ⟹  8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
          ⟹ 13
```

`(vector-set!` *vector k obj*`)` procedure

K must be a valid index of *vector*. The `vector-set!` procedure stores *obj* in element *k* of *vector*. The value returned by `vector-set!` is the unspecified value.

Passing an immutable vector to `vector-set!` should cause an exception with condition type `&contract` to be raised.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
          ⟹  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
          ⟹  unspecified
             ; constant vector
             ; may raise &contract exception
```

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

The `vector->list` procedure returns a newly allocated list of the objects contained in the elements of *vector*. The `list->vector` procedure returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '#(dah dah didah))
          ⟹  (dah dah didah)
(list->vector '(dididit dah))
          ⟹  #(dididit dah)
```

`(vector-fill!` *vector fill*`)` procedure

Stores *fill* in every element of *vector* and returns the unspecified value.

9.17. Errors and violations

`(error` *who message irritant*$_1$ ...`)` procedure
`(contract-violation` *who message irritant*$_1$ ...`)` procedure

Who must be a string or a symbol or `#f`. *message* must be a string. The *irritant*s are arbitrary objects.

These procedures raise an exception. Calling the `error` procedure means that an error has occurred, typically caused by something that has gone wrong in the interaction of the program with the external world or the user. Calling the `contract-violation` procedure means that an invalid call to a procedure was made, either passing an invalid number of arguments, or passing an argument that it is not specified to handle.

The *who* argument should describe the procedure or operation that detected the exception. The *message* argument should describe the exceptional situation. The *irritant*s

should be the arguments to the operation that detected the operation.

The condition object provided with the exception (see chapter 14) has the following condition types:

- If *who* is not `#f`, the condition has condition type `&who`, with *who* as the value of the `who` field. In that case, *who* should identify the procedure or entity that detected the exception. If it is `#f`, the condition does not have condition type `&who`.
- The condition has condition type `&message`, with *message* as the value of the `message` field.
- The condition has condition type `&irritants`, and the `irritants` field has as its value a list of the *irritant*s.

Moreover, the condition created by `error` has condition type `&error`, and the condition created by `contract-violation` has condition type `&contract`.

```
(define (fac n)
  (if (not (integer-valued? n))
      (contract-violation
       'fac "non-integral argument" n))
  (if (negative? n)
      (contract-violation
       'fac "negative argument" n))
  (letrec
    ((loop (lambda (n r)
             (if (zero? n)
                 r
                 (loop (- n 1) (* r n))))))
      (loop n 1)))

(fac 5)                    ⟹ 120
(fac 4.5)                  ⟹ &contract exception
(fac -3)                   ⟹ &contract exception
```

Rationale: The procedures encode a common pattern of raising exceptions.

9.18. Control features

This chapter describes various primitive procedures which control the flow of program execution in special ways.

`(apply` *proc* arg_1 ... *args*`)` procedure

Proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list `(append (list` arg_1 ... `)` *args*`)` as the actual arguments.

```
(apply + (list 3 4))       ⟹ 7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)   ⟹ 30
```

`(call-with-current-continuation` *proc*`)` procedure
`(call/cc` *proc*`)` procedure

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

The escape procedure accepts the same number of arguments as the continuation of the original call to `call-with-current-continuation`.

The escape procedure that is passed to *proc* has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

The following examples show only the most common ways in which `call-with-current-continuation` is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of `call-with-current-continuation`.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                   ⟹ -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))   ⟹ 4

(list-length '(a b . c))   ⟹ #f
(call-with-current-continuation procedure?)
                           ⟹ #t
```

Rationale:

A common use of `call-with-current-continuation` is for structured, non-local exits from loops or procedure bodies, but in fact `call-with-current-continuation` is extremely useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a *continuation* wanting the result of the expression. The continuation represents an entire (default) future for the computation. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and store the result in some other variable. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. The `call-with-current-continuation` procedure allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [33] invented a general purpose escape operator called the J-operator. John Reynolds [43] described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the `catch` construct could be provided by a procedure instead of by a special syntactic construct, and the name `call-with-current-continuation` was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name `call/cc` instead.

`(values` *obj* ...`)` procedure

Delivers all of its arguments to its continuation. The `values` procedure might be defined as follows:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

The continuations of all non-final expressions within a sequence of expressions in `lambda`, `begin`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `case`, `cond`, and `do` forms as well as the continuations of the *before* and *after* arguments to `dynamic-wind` take an arbitrary number of values.

Except for these and the continuations created by the `call-with-values` procedure, all other continuations take exactly one value. The effect of passing an inappropriate number of values to a continuation not created by `call-with-values` is undefined.

`(call-with-values` *producer consumer*`)` procedure

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

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                              ⟹  5

(call-with-values * -)        ⟹  -1
```

If an inappropriate number of values is passed to a continuation created by `call-with-values`, an exception with condition type `&contract` is raised.

`(dynamic-wind` *before thunk after*`)` procedure

Before, *thunk*, and *after* must be procedures accepting zero arguments and returning any number of values.

In the absense of any calls to escape procedures (see `call-with-current-continuation`), `dynamic-wind` behaves as if defined as follows.

```
(define dynamic-wind
  (lambda (before thunk after)
    (before)
    (call-with-values
      (lambda () (thunk))
      (lambda vals
        (after)
        (apply values vals)))))
```

That is, *before* is called without arguments. If *before* returns, *thunk* is called without arguments. If *thunk* returns, *after* is called without arguments. Finally, if *after* returns, the values resulting from the call to *thunk* are returned.

Invoking an escape procedure to transfer control into or out of the dynamic extent of the call to *thunk* can cause additional calls to *before* and *after*. When an escape procedure created outside the dynamic extent of the call to *thunk* is invoked from within the dynamic extent, *after* is called just after control leaves the dynamic extent. Similarly, when an escape procedure created within the dynamic extent of the call to *thunk* is invoked from outside the dynamic extent, *before* is called just before control reenters the dynamic extent. In the latter case, if *thunk* returns, *after* is called even if *thunk* has returned previously. While the calls to *before* and *after* are not considered to be within the dynamic extent of the call to *thunk*, calls to the before and after thunks of any other calls to `dynamic-wind` that occur within the dynamic extent of the call to *thunk* are considered to be within the dynamic extent of the call to *thunk*.

More precisely, an escape procedure used to transfer control out of the dynamic extent of a set of zero or more active `dynamic-wind` *thunk* calls x ... and transfer control into the dynamic extent of a set of zero or more active `dynamic-wind` *thunk* calls y ... proceeds as follows. It leaves the dynamic extent of the most recent x and calls

without arguments the corresponding *after* thunk. If the *after* thunk returns, the escape procedure proceeds to the next most recent x, and so on. Once each x has been handled in this manner, the escape procedure calls without arguments the *before* thunk corresponding to the least recent y. If the *before* thunk returns, the escape procedure reenters the dynamic extent of the least recent y and proceeds with the next least recent y, and so on. Once each y has been handled in this manner, control is transfered to the continuation packaged in the escape procedure.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

          ⟹ (connect talk1 disconnect
               connect talk2 disconnect)

(let ((n 0))
  (call-with-current-continuation
    (lambda (k)
      (dynamic-wind
        (lambda ()
          (set! n (+ n 1))
          (k))
        (lambda ()
          (set! n (+ n 2)))
        (lambda ()
          (set! n (+ n 4))))))
  n)                          ⟹ 1

(let ((n 0))
  (call-with-current-continuation
    (lambda (k)
      (dynamic-wind
        values
        (lambda ()
          (dynamic-wind
            values
            (lambda ()
              (set! n (+ n 1))
              (k))
            (lambda ()
              (set! n (+ n 2))
              (k))))
        (lambda ()
          (set! n (+ n 4))))))
  n)                          ⟹ 7
```

9.19. Iteration

`(let` ⟨variable⟩ ⟨bindings⟩ ⟨body⟩`)` syntax

"Named `let`" is a variant on the syntax of `let` which provides a more general looping construct than `do` and may also be used to express recursions. It has the same syntax and semantics as ordinary `let` except that ⟨variable⟩ is bound within ⟨body⟩ to a procedure whose formal arguments are the bound variables and whose body is ⟨body⟩. Thus the execution of ⟨body⟩ may be repeated by invoking the procedure named by ⟨variable⟩.

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))
          ⟹  ((6 1 3) (-5 -2))
```

The `let` keyword could be defined in terms of `lambda` and `letrec` using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
        tag)
      val ...))))
```

`(do (((`⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩`)` syntax
`...)`
`(`⟨test⟩ ⟨expression⟩ `...)`
⟨expressionx⟩ `...)`

Syntax: The ⟨init⟩s, ⟨step⟩s, and ⟨test⟩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. When a termination condition is met, the loop exits after evaluating the ⟨expression⟩s.

A `do` expression are 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 false (see section 9.11), then the ⟨command⟩ expressions are evaluated in order for effect, the ⟨step⟩ expressions are evaluated in some unspecified order, the ⟨variable⟩s are bound to fresh locations, the results of the ⟨step⟩s are stored in the bindings of the ⟨variable⟩s, and the next iteration begins.

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

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

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

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))    ⟹  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))     ⟹  25
```

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

```
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...)
         (test expr ...)
         command ...)
     (letrec
       ((loop
         (lambda (var ...)
           (if test
               (begin
                 (unspecified)
                 expr ...)
               (begin
                 command
                 ...
                 (loop (do "step" var step ...)
                       ...)))))))
       (loop init ...)))
    ((do "step" x)
     x)
    ((do "step" x y)
     y)))
```

sectionQuasiquotation

`(quasiquote` ⟨qq template⟩`)` syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance. If no `unquote` or `unquote-splicing` forms appear within the ⟨qq template⟩, the result of evaluating `(quasiquote` ⟨qq template⟩`)` is equivalent to the result of evaluating `(quote` ⟨qq template⟩`)`.

If an `(unquote` ⟨expression⟩ ...`)` form appears inside a ⟨qq template⟩, however, the ⟨expression⟩s are evaluated ("unquoted") and their results are inserted into the structure instead of the `unquote` form.

If an `(unquote-splicing` ⟨expression⟩ ...`)` form appears inside a ⟨qq template⟩, then the ⟨expression⟩s must evaluate to lists; the opening and closing parentheses of the list are then "stripped away" and the elements of the lists are inserted in place of the `unquote-splicing` form.

`unquote-splicing` and multi-operand `unquote` forms must appear only within a list or vector ⟨qq template⟩.

As noted in section 3.3.5, `(quasiquote` ⟨qq template⟩`)` may be abbreviated `` `⟨qq template⟩ ``, `(unquote` ⟨expression⟩`)` may be abbreviated `,`⟨expression⟩, and `(unquote-splicing` ⟨expression⟩`)` may be abbreviated `,@`⟨expression⟩.

```
`(list ,(+ 1 2) 4)          ⟹  (list 3 4)
(let ((name 'a)) `(list ,name ',name))
         ⟹  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
         ⟹  (a 3 4 5 6 b)
`(( foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
         ⟹  ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
         ⟹  #(10 5 2 4 3 8)
(let ((name 'foo))
  `((unquote name name name)))
         ⟹ (foo foo foo)
(let ((name '(foo)))
  `((unquote-splicing name name name)))
         ⟹ (foo foo foo)
(let ((q '((append x y) (sqrt 9))))
  ``(foo ,,@q))
         ⟹ `(foo (unquote (append x y) (sqrt 9)))
(let ((x '(2 3))
      (y '(4 5)))
  `(foo (unquote (append x y) (sqrt 9))))
         ⟹ (foo (2 3 4 5) 3)
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost `quasiquote`. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
         ⟹  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
         ⟹  (a `(b ,x ,'y d) e)
```

It is a syntax violation if any of the identifiers `quasiquote`, `unquote`, or `unquote-splicing` appear in positions within a ⟨qq template⟩ otherwise than as described above.

The following grammar for quasiquote expressions is not context-free. It is presented as a recipe for generating an infinite number of production rules. Imagine a copy of the following rules for $D = 1, 2, 3, \ldots$. D keeps track of the nesting depth.

⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨qq template 0⟩ ⟶ ⟨expression⟩
⟨quasiquotation D⟩ ⟶ `(quasiquote` ⟨qq template D⟩`)`
⟨qq template D⟩ ⟶ ⟨simple datum⟩
  | ⟨list qq template D⟩
  | ⟨vector qq template D⟩
  | ⟨unquotation D⟩
⟨list qq template D⟩ ⟶ `(`⟨qq template or splice D⟩`*)`
  | `(`⟨qq template or splice D⟩$^+$ `.` ⟨qq template D⟩`)`
  | ⟨quasiquotation $D+1$⟩
⟨vector qq template D⟩ ⟶ `#(`⟨qq template or splice D⟩`*)`
⟨unquotation D⟩ ⟶ `(unquote` ⟨qq template $D-1$⟩`)`
⟨qq template or splice D⟩ ⟶ ⟨qq template D⟩
  | ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶
    `(unquote-splicing` ⟨qq template $D-1$⟩`*)`
  | `(unquote` ⟨qq template $D-1$⟩`*)`

In ⟨quasiquotation⟩s, a ⟨list qq template D⟩ can sometimes be confused with either an ⟨unquotation D⟩ or a ⟨splicing unquotation D⟩. The interpretation as an ⟨unquotation⟩ or ⟨splicing unquotation D⟩ takes precedence.

9.20. Binding constructs for syntactic keywords

The `let-syntax` and `letrec-syntax` forms are analogous to `let` and `letrec` but bind keywords rather than variables. Like `begin`, a `let-syntax` or `letrec-syntax` form may appear in a definition context, in which case it is treated as a definition, and the forms in the body of the form must also be definitions. A `let-syntax` or `letrec-syntax` form may also appear in an expression context, in which case the forms within their bodies must be expressions.

`(let-syntax` ⟨bindings⟩ ⟨form⟩ `...)` syntax

Syntax: ⟨Bindings⟩ must have the form

`((`⟨keyword⟩ ⟨transformer spec⟩`) ...)`

Each ⟨keyword⟩ is an identifier, each ⟨transformer spec⟩ is either an instance of `syntax-rules` or an expression that evaluates to a transformer (see chapter 17). It is a syntax violation for ⟨keyword⟩ to appear more than once in the list of keywords being bound. The ⟨form⟩s are arbitrary forms.

Semantics: The ⟨form⟩s are expanded in the syntactic environment obtained by extending the syntactic environment of the `let-syntax` form with macros whose keywords are the ⟨keyword⟩s, bound to the specified transformers. Each binding of a ⟨keyword⟩ has the ⟨form⟩s as its region.

The ⟨form⟩s of a `let-syntax` form are treated, whether in definition or expression context, as if wrapped in an implicit `begin`, see section 9.5.7. Thus, internal definitions in the result of expanding the ⟨form⟩s have the same region as any definition appearing in place of the `let-syntax` form would have.

```
(let-syntax ((when (syntax-rules ()
                     ((when test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now))
    if))                        ⟹  now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m))))                    ⟹  outer
(let ()
  (let-syntax
    ((def (syntax-rules ()
            ((def stuff ...) (define stuff ...)))))
    (def foo 42))
  foo)                          ⟹ 42

(let ()
  (let-syntax ())
  5)                            ⟹ 5
```

`(letrec-syntax` ⟨bindings⟩ ⟨form⟩ `...)` syntax

Syntax: Same as for `let-syntax`.

Semantics: The ⟨form⟩s are expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` form with macros whose keywords are the ⟨keyword⟩s, bound to the specified transformers. Each binding of a ⟨keyword⟩ has the ⟨bindings⟩ as well as the ⟨form⟩s within its region, so the transformers can transcribe forms into uses of the macros introduced by the `letrec-syntax` form.

The ⟨form⟩s of a `letrec-syntax` form are treated, whether in definition or expression context, as if wrapped in an implicit `begin`, see section 9.5.7. Thus, internal definitions in the result of expanding the ⟨form⟩s have the same region as any definition appearing in place of the `letrec-syntax` form would have.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))              ⟹  7
```

The following example highlights how `let-syntax` and `letrec-syntax` differ.

```
(let ((f (lambda (x) (+ x 1))))
  (let-syntax ((f (syntax-rules ()
                    ((f x) x)))
               (g (syntax-rules ()
                    ((g x) (f x)))))
    (list (f 1) (g 1))))
      ⟹ (1 2)

(let ((f (lambda (x) (+ x 1))))
  (letrec-syntax ((f (syntax-rules ()
                       ((f x) x)))
                  (g (syntax-rules ()
                       ((g x) (f x)))))
    (list (f 1) (g 1))))
      ⟹ (1 1)
```

The two expressions are identical except that the `let-syntax` form in the first expression is a `letrec-syntax` form in the second. In the first expression, the `f` occurring in `g` refers to the `let`-bound variable `f`, whereas in the second it refers to the keyword `f` whose binding is established by the `letrec-syntax` form.

9.21. `syntax-rules`

A ⟨transformer spec⟩ in a syntax definition, `let-syntax`, or `letrec-syntax` can be a `syntax-rules` form:

`(syntax-rules (`⟨literal⟩ `...)` ⟨syntax rule⟩ `...)` syntax

Syntax: Each ⟨literal⟩ must be an identifier. Each ⟨syntax rule⟩ must take one of the the following forms:

(⟨srpattern⟩ ⟨template⟩)
(⟨srpattern⟩ ⟨fender⟩ ⟨template⟩)

An ⟨srpattern⟩ is a restricted form of ⟨pattern⟩, namely, a nonempty ⟨pattern⟩ in one of four parenthesized forms below whose first subform is an identifier or an underscore _. A ⟨pattern⟩ is an identifier, constant, or one of the following.

(⟨pattern⟩ ...)
(⟨pattern⟩ ⟨pattern⟩ ... . ⟨pattern⟩)
(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ...)
(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ... . ⟨pattern⟩)
#(⟨pattern⟩ ...)
#(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ...)

An ⟨ellipsis⟩ is the identifier "`...`" (three periods).

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

(⟨subtemplate⟩ ...)
(⟨subtemplate⟩ ... . ⟨template⟩)
#(⟨subtemplate⟩ ...)

A ⟨subtemplate⟩ is a ⟨template⟩ followed by zero or more ellipses.

⟨Fender⟩, if present, is an expression. The ⟨fender⟩ is generally useful only if the `(r6rs syntax-case)` library has been imported; see chapter 17.

Semantics: An instance of `syntax-rules` produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the ⟨syntax rule⟩s, beginning with the leftmost ⟨syntax rule⟩. When a match is found, the macro use is transcribed hygienically according to the template. It is a syntax violation when no match is found.

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⟩ ...).

While the first subform of ⟨srpattern⟩ may be an identifier, the identifier is not involved in the matching and is not considered a pattern variable or literal identifier.

Rationale: The identifier is most often the keyword used to identify the macro. The scope of the keyword is determined by the binding form or syntax definition that binds it to the associated macro transformer. If the keyword were a pattern variable or literal identifier, then the template that follows the pattern would be within its scope regardless of whether the keyword were bound by `let-syntax`, `letrec-syntax`, or `define-syntax`.

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 may 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 identifier such that both P and F would refer to the same binding if both were to appear in the output of the macro outside of any bindings inserted into the output of the macro. (If neither of two like-named identifiers refers to any binding, i.e., both are undefined, they are considered to refer to the same binding.)
- P is of the form `(`P_1 `...` P_n`)` and F is a list of n elements that match P_1 through P_n.
- P is of the form `(`P_1 `...` P_n `.` P_x`)` and F is a list or improper list of n or more elements whose first n elements match P_1 through P_n and whose nth cdr matches P_x.
- P is of the form `(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a proper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.
- P is of the form `(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n `.` P_x`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a list or improper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, whose next $n-m$ elements match P_{m+1} through P_n, and whose nth and final cdr matches P_x.
- P is of the form `#(`P_1 `...` P_n`)` and F is a vector of n elements that match P_1 through P_n.
- P is of the form `#(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a vector of n or more elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.
- P is a pattern datum (any nonlist, nonvector, nonsymbol datum) and F is equal to P in the sense of the `equal?` procedure.

When a macro use is transcribed according to the template of the matching ⟨syntax rule⟩, pattern variables that occur in the template are replaced by the subforms they match in the input.

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 consraints of this paragraph are not met.

A template of the form `(`⟨ellipsis⟩ ⟨template⟩`)` is identical to ⟨template⟩, except that ellipses within the template have no special meaning. That is, any ellipses contained within ⟨template⟩ are treated as ordinary identifiers. In particular, the template `(... ...)` produces a single ellipsis, `...`. This allows syntactic abstractions to expand into forms containing ellipses.

As an example, if `let` and `cond` are defined as in section 9.5.6 and appendix B then they are hygienic (as required) and the following is not an error.

```
(let ((=> #f))
  (cond (#t => 'ok)))       ⟹ ok
```

The macro transformer for `cond` recognizes `=>` as a local variable, and hence an expression, and not as the top-level identifier `=>`, which the macro transformer treats as a syntactic keyword. Thus the example expands into

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

instead of

```
(let ((=> #f))
  (let ((temp #t))
    (if temp ('ok temp))))
```

which would result in a contract violation.

9.22. Declarations

A declaration affects a range of code and indicates that the code within that range should be compiled or executed to have certain qualities. A declaration appears at the beginning of a ⟨body⟩ (see section 9.4), at the beginning of a ⟨library body⟩ (see chapter 6), or within a ⟨script body⟩ (see chapter 7), and its range is the body in which it appears, possibly but not necessarily including code inserted into the range by macro expansion.

A ⟨declaration⟩ is always a `declare` form.

`(declare` ⟨declare spec⟩*`)` syntax

A ⟨declare spec⟩ has one of the following forms:

- `(`⟨quality⟩ ⟨priority⟩`)`

  ⟨Quality⟩ has to be one of `safe`, `fast`, `small`, and `debug`. ⟨priority⟩ has to be one of `0`, `1`, `2`, and `3`.

  This specifies that the code in the range of the declaration should have the indicated quality at the indicated priority, where priority 3 is the highest priority. Priority 0 means the quality is not a priority at all.

- ⟨quality⟩

  This is a synonym for `(`⟨quality⟩ `3)`.

- `unsafe`

  This is a synonym for `(safe 0)`.

For `safe`, the default priority must be 1 or higher. When the priority for `safe` is 1 or higher, implementations must raise all required exceptions and let them be handled by the exception mechanism (see chapter 14).

Beyond that, the detailed interpretation of declarations will vary in different implementations. In particular, implementations are free to ignore declarations, and may observe some declarations while ignoring others.

The following descriptions of each quality may provide some guidance for programmers and implementors.

`safe` This quality's priority influences the degree of checking for exceptional situations, and the raising and handling of exceptions in response to those situations. The higher the priority, the more likely an exception is raised.

At priority 0, an implementation is allowed to ignore any requirements for raising an exception with condition type `&violation` (or one of its subtypes). In situations for which this report allows or requires the implementation to raise an exception with condition type `&violation`, the implementation may ignore the situation and continue the computation with an incorrect result, may terminate the computation in an unpleasant fashion, or may destroy the invariants of run-time data structures in ways that cause unexpected and mysterious misbehavior even in code that comes within the scope of a safe declaration. All bets are off.

At priority 1 and higher, an implementation must raise all exceptions required by this report, handle those exceptions using the exception mechanism described in chapter 14, and use the default exception handlers described in that chapter. See also section 4.4.

At higher priorities, implementations may be more likely to raise exceptions that are allowed but not required by this report.

Most implementations are able to recognize some violations when parsing, expanding macros, or compiling a definition or expression whose evaluation has not yet commenced in the usual sense. Implementations are allowed to use nonstandard exception handlers at those times, and are encouraged to raise `&syntax` exceptions for violations detected at those times, even if the definition or expression that contains the violation will never be executed. Implementations are also allowed to raise a `&warning` exception at those times if they determine that some subexpression would inevitably raise some kind of `&violation` exception were it ever to be evaluated.

`fast` This quality's priority influences the speed of the code it governs. At high priorities, the code is likely to run faster, but that improvement is constrained by other qualities and may come at the expense of the small and debug qualities.

`small` This quality's priority influences the amount of computer memory needed to represent and to run the code. At high priorities, the code is likely to occupy less memory and to require less memory during evaluation.

`debug` This quality's priority influences the programmer's ability to debug the code. At high priorities, the programmer is more likely to understand the correspondence between the original source code and information displayed by debugging tools. At low priorities, some debugging tools may not be usable.

9.23. Tail calls and tail contexts

A *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined inductively. Note that a tail context is always determined with respect to a particular lambda expression.

- The last expression within the body of a lambda expression, shown as ⟨tail expression⟩ below, occurs in a tail context.

  `(lambda` ⟨formals⟩
  ⟨declaration⟩* ⟨definition⟩*
  ⟨expression⟩* ⟨tail expression⟩`)`

- If one of the following expressions is in a tail context, then the subexpressions shown as ⟨tail expression⟩ are in a tail context. These were derived from rules for the syntax of the forms described in this chapter by replacing some occurrences of ⟨expression⟩ with ⟨tail expression⟩. Only those rules that contain tail contexts are shown here.

  `(if` ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩`)`
  `(if` ⟨expression⟩ ⟨tail expression⟩`)`

  `(cond` ⟨cond clause⟩$^+$`)`
  `(cond` ⟨cond clause⟩* `(else` ⟨tail sequence⟩`))`

  `(case` ⟨expression⟩
  ⟨case clause⟩$^+$`)`
  `(case` ⟨expression⟩
  ⟨case clause⟩*
  `(else` ⟨tail sequence⟩`))`

  `(and` ⟨expression⟩* ⟨tail expression⟩`)`
  `(or` ⟨expression⟩* ⟨tail expression⟩`)`

  `(let (`⟨binding spec⟩*`)` ⟨tail body⟩`)`
  `(let` ⟨variable⟩ `(`⟨binding spec⟩*`)` ⟨tail body⟩`)`
  `(let* (`⟨binding spec⟩*`)` ⟨tail body⟩`)`
  `(letrec* (`⟨binding spec⟩*`)` ⟨tail body⟩`)`
  `(letrec (`⟨binding spec⟩*`)` ⟨tail body⟩`)`
  `(let-values (`⟨mv binding spec⟩*`)` ⟨tail body⟩`)`
  `(let*-values (`⟨mv binding spec⟩*`)` ⟨tail body⟩`)`

  `(let-syntax (`⟨syntax spec⟩*`)` ⟨tail body⟩`)`
  `(letrec-syntax (`⟨syntax spec⟩*`)` ⟨tail body⟩`)`

  `(begin` ⟨tail sequence⟩`)`

  `(do (`⟨iteration spec⟩*`)`
  `(`⟨test⟩ ⟨tail sequence⟩`)`
  ⟨expression⟩*`)`

  where

  ⟨cond clause⟩ ⟶ `(`⟨test⟩ ⟨tail sequence⟩`)`
  ⟨case clause⟩ ⟶ `((`⟨datum⟩*`)` ⟨tail sequence⟩`)`

  ⟨tail body⟩ ⟶ ⟨declaration⟩* ⟨definition⟩*
  ⟨tail sequence⟩
  ⟨tail sequence⟩ ⟶ ⟨expression⟩* ⟨tail expression⟩

- If a `cond` expression is in a tail context, and has a clause of the form `(`⟨expression$_1$⟩ `=>` ⟨expression$_2$⟩`)` then the (implied) call to the procedure that results from the evaluation of ⟨expression$_2$⟩ is in a tail context. ⟨expression$_2$⟩ itself is not in a tail context.

Certain built-in procedures are also required to perform tail calls. The first argument passed to `apply` and to `call-with-current-continuation`, and the second argument passed to `call-with-values`, must be called via a tail call.

In the following example the only tail call is the call to `f`. None of the calls to `g` or `h` are tail calls. The reference to `x` is in a tail context, but it is not a call and thus is not a tail call.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

Note: Implementations are allowed, but not required, to recognize that some non-tail calls, such as the call to `h` above, can be evaluated as though they were tail calls. In the example above, the `let` expression could be compiled as a tail call to `h`. (The possibility of `h` returning an unexpected number of values can be ignored, because in that case the effect of the `let` is explicitly unspecified and implementation-dependent.)

DESCRIPTION OF THE STANDARD LIBRARIES

10. Unicode

The procedures exported by the `(r6rs unicode)` library provide access to some aspects of the Unicode semantics for characters and strings: category information, case-independent comparisons, case mappings, and normalization [51].

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

10.1. Characters

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

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

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

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

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

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

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

Note that these character-based procedures are an incomplete approximation to case conversion, even ignoring the user's locale. In general, case mappings require the context of a string, both in arguments and in result. The `string-upcase`, `string-downcase`, `string-titlecase`, and `string-foldcase` procedures (section 10.2) perform more general case conversion.

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

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

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

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

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

A character is alphabetic if it is a Unicode letter, i.e. if it is in one of the categories Lu, Ll, Lt, Lm, and Lo. A character is numeric if it is in categeory Nd. A character is whitespace if it is in one of the space, line, or paragraph separator categories (Zs, Zl or Zp), or if is Unicode 9 (Horizontal tabulation), Unicode 10 (Line feed), Unicode 11 (Vertical tabulation), Unicode 12 (Form feed), or Unicode 13 (Carriage return). A character is upper case if it has the Unicode "Uppercase" property, lower case if it has the "Lowercase" property, and title case if it is in the Lt general category.

```
(char-alphabetic? #\a)     ⟹ #t
(char-numeric? #\1)        ⟹ #t
```

```
(char-whitespace? #\space)  ⟹ #t
(char-whitespace? #\x00A0)  ⟹ #t
(char-upper-case? #\Σ)      ⟹ #t
(char-lower-case? #\σ)      ⟹ #t
(char-lower-case? #\x00AA)  ⟹ #t
(char-title-case? #\I)      ⟹ #f
(char-title-case? #\x01C5)  ⟹ #t
```

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

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

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

10.2. 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 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.

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

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

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

(string-downcase "Σ")       ⟹ "σ"

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

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

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

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

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

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

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

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

These procedures take a string argument and return a string result, which is the input string normalized to Unicode normalization form D, KD, C, or KC, respectively.

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

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

11. Bytes objects

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

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

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

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

Like list and vector literals, literals representing bytes objects must be quoted:

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

`(endianness big)` syntax
`(endianness little)` syntax

`(endianness big)` and `(endianness little)` evaluate to the symbols `big` and `little`, respectively. These symbols represent an endianness, and whenever one of the procedures operating on bytes objects accepts an endianness as an argument, that argument must be one of these symbols. It is a syntax violation for the operand to `endianness` to be anything other than `big` or `little`.

`(native-endianness)` procedure

Returns the implementation's preferred endianness (usually that of the underlying machine architecture), either `big` or `little`.

`(bytes? `*obj*`)` procedure

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

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

Returns a newly allocated bytes object of *k* bytes.

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

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

`(bytes-length `*bytes*`)` procedure

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

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

K must be a valid index of bytes.

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

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

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

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

K must be a valid index of *bytes*.

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

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

Both procedures return the unspecified value.

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

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

  (list
    (bytes-s8-ref b 0)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(bytes-u16-native-set! b 0 12345)
(bytes-u16-native-ref b 0)  ⟹ 12345

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

`(bytes-u32-ref` *bytes k endianness*`)` procedure
`(bytes-s32-ref` *bytes k endianness*`)` procedure

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`(bytes=?` *bytes*$_1$ *bytes*$_2$) procedure

Returns `#t` if *bytes*$_1$ and *bytes*$_2$ are equal—that is, if they have the same length and equal bytes at all valid indices. It returns `#f` otherwise.

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

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

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

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

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

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

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

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

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

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

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

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

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

`(bytes-copy!` *source source-start target target-start n*`)` procedure

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

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

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

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

$$\{\textit{source-start}, \ldots \textit{source-start} + n - 1\}$$

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

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

This returns the unspecified value.

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

`(bytes-copy` *bytes*`)` procedure

Returns a newly allocated copy of *bytes*.

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

List must be a list of octets.

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

The `u8-list->bytes` procedure returns a newly allocated bytes object whose elements are the elements of list *list*, in the same order. Analogous to `list->vector`.

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

Size must be a positive exact integer.

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

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

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

12. List utilities

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

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

Proc must be a procedure; it must take a single argument if *list* is non-empty. The `find` procedure applies *proc* to the elements of *list* in order. If *proc* returns a true value for an element, `find` immediately returns that element. If *proc* returns `#f` for all elements of the list, it returns `#f`.

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

`(forall` *proc* l_1 l_2 ...`)` procedure
`(exists` *proc* l_1 l_2 ...`)` procedure

The ls must all be the empty list, or chains of pairs of sizes according to the condition specified below. *proc* must be a procedure; it must take a single argument if the ls are non-empty.

The `forall` procedure applies *proc* element-wise to the elements of the ls. If *proc* returns `#t` for all but the last elements of the ls, *forall* performs a tail call of *proc* on the last elements—in this case, the ls must all be lists. If *proc* returns `#f` on any set of elements, `forall` returns `#f` after the first such application of *proc* without further traversing the ls. If the ls are all empty, `forall` returns `#t`.

The `exists` procedure applies *proc* element-wise to the elements of the ls. If *proc* returns `#f` for all but the last elements of the ls, *exists* performs a tail call of *proc* on the last elements—in this case, the ls must all be lists. If *proc* returns a true value on any set of elements, `exists` returns that value after the first such application of *proc* without further traversing the ls. If the ls are all empty, `exists` returns `#f`.

```
(forall even? '(3 1 4 1 5 9)) ⟹ #f
(forall even? '(3 1 4 1 5 9 2)) ⟹ #f
(forall even? '(2 1 4 14))  ⟹ #t
(forall even? '(2 1 4 14 . 9))
          ⟹ &contract exception
(forall (lambda (n) (and (even? n) n)) '(2 1 4 14)
          ⟹ 14

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

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

Proc must be a procedure; it must take a single argument if *list* is non-empty. The `filter` procedure successively applies *proc* to the elements of *list* and returns a list of the values of *list* for which *proc* returned a true value. The `partition` procedure also successively applies *proc* to the elements of *list*, but returns two values, the first one a list of the values of *list* for which *proc* returned a true value, and the second a list of the values of *list* for which *proc* returned `#f`.

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

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

(`fold-left` *kons nil* $list_1$ $list_2$ ... $list_n$) procedure

If more than one *list* is given, then they must all be the same length. *kons* must be a procedure; if the *list*s are non-empty, it must take one more argument than there are *list*s. The `fold-left` procedure iterates the *kons* procedure over an accumulator value and the values of the *list*s from left to right, starting with an accumulator value of *nil*. More specifically, `fold-left` returns *nil* if the *list*s are empty. If they are not empty, *kons* is first applied to *nil* and the respective first elements of the *list*s in order. The result becomes the new accumulator value, and *kons* is applied to new accumulator value and the respective next elements of the *list*. This step is repeated until the end of the list is reached; then the accumulator value is returned.

```
(fold-left + 0 '(1 2 3 4 5)) ⟹ 15

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

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

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

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

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

(`fold-right` *kons nil* $list_1$ $list_2$ ... $list_n$) procedure

If more than one *list* is given, then they must all be the same length. *kons* must be a procedure; if the *list*s are non-empty, it must take one more argument than there are *list*s. The `fold-right` procedure iterates the *kons* procedure over the values of the *list*s from right to left and an accumulator value, starting with an accumulator value of *nil*. More specifically, `fold-right` returns *nil* if the *list*s are empty. If they are not empty, *kons* is first applied to the respective last elements of the *list*s in order and *nil*. The result becomes the new accumulator value, and *kons* is applied to the respective previous elements of the *list* and the new accumulator value. This step is repeated until the beginning of the list is reached; then the accumulator value is returned.

```
(fold-right + 0 '(1 2 3 4 5)) ⟹ 15

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

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

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

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

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

(`remv` *obj list*) procedure
(`remq` *obj list*) procedure

Proc must be a procedure; it must take a single argument if *list* is non-empty. Each of these procedures returns a list of the elements of *list* that do not satisfy a given condition. The `remp` procedure successively applies *proc* to the elements of *list* and returns a list of the values of *list* for which *proc* returned `#f`. The `remove`, `remv`, and `remq` procedures return a list of the elements that are not *obj*. The `remq` procedure uses `eq?` to compare *obj* with the elements of *list*, while `remv` uses `eqv?` and `remove` uses `equal?`.

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

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

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

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

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

Proc must be a procedure; it must take a single argument if *l* is non-empty. *l* must be the empty list or a chain of pairs of size according to the conditions stated below.

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

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

(memq 'a '(a b c))          ⟹  (a b c)
(memq 'b '(a b c))          ⟹  (b c)
(memq 'a '(b c d))          ⟹  #f
(memq (list 'a) '(b (a) c)) ⟹  #f
(member (list 'a)
        '(b (a) c))         ⟹  ((a) c)
(memq 101 '(100 101 102))   ⟹  unspecified
(memv 101 '(100 101 102))   ⟹  (101 102)
```

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

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

Al (for "association list") must be the empty list or a chain of pairs where each car contains a pair of size according to the conditions specified below. *Proc* must be a procedure; it must take a single argument if *al* is non-empty.

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

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

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

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

13. Records

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

- the `(r6rs records procedural)` library, a procedural layer for creating and manipulating record types and record instances,
- the `(r6rs records explicit)` library, an explicit-naming syntactic layer for defining record types and explicitly named bindings for various procedures to manipulate the record type,

- the `(r6rs records implicit)` library, an implicit-naming syntactic layer that extends the explicit-naming syntactic layer, allowing the names of the defined procedures to be determined implicitly from the names of the record type and fields, and
- the `(r6rs records inspection)` library, a set of inspection procedures.

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

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

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

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

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

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

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

This section uses the *rtd* and *constructor-descriptor* parameter names for arguments that must be record-type descriptors and constructor descriptors, respectively (see section 13.1).

13.1. Procedural layer

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

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

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

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

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

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

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

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

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

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

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

The *fields* argument must be a list of field specifiers. Each field specifier must be a list of the form `(mutable` *name*`)` or a list of the form `(immutable` *name*`)`. Each name must be a symbol and names the corresponding field of the record type; the names need not be distinct. A field identified as mutable may be modified, whereas an attempt to obtain a mutator for a field identified as immutable raises an exception with condition type `&contract`. 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.

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

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

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

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

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

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

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

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

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

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

Returns a *record-constructor descriptor* (or *constructor descriptor* for short) that can be used to create record constructors (via `record-constructor`; see below) or other constructor descriptors. *rtd* must be a record-type descriptor. *protocol* must be a procedure or `#f`. If it is `#f`, a default *protocol* procedure is supplied. If *protocol* is a procedure, it is called by `record-constructor` with a single argument p and must return a procedure that creates and returns an instance of the record type using p as described below.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13.2. Explicit-naming syntactic layer

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

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

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

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

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

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

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

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

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

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

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

- `(fields` ⟨field-spec⟩*`)`

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

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

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

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

- `(parent` ⟨parent name⟩`)`

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

- `(protocol` ⟨expression⟩`)`

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

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

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

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

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

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

- `(nongenerative` ⟨uid⟩`)`

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

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

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

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

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

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

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

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

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

Explicit-naming syntactic-layer examples:

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

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

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

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

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

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

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

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

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

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

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

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

13.3. Implicit-naming syntactic layer

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

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

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

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

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

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

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

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

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

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

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

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

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


(define *ex3-instance* #f)

(define-record-type ex3
  (parent cpoint)
  (protocol
   (lambda (p)
```

```
    (lambda (x y t)
      (let ((r ((p x y 'red) t)))
        (set! *ex3-instance* r)
        r))))
  (fields
   (mutable thickness))
  (sealed #t) (opaque #t))

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

(record? ex3-i1)            ⟹ #f
```

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

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

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

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

13.4. Inspection

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Returns a list 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`.

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

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

14. Exceptions and conditions

Scheme allows programs to deal with exceptional situations using two cooperating facilities: The exception system allows the program, when it detects an exceptional situation, to pass control to an exception handler, and for dynamically establishing such exception handlers. Exception handlers are always invoked with an object describing the exceptional situation. Scheme's condition system provides a standardized taxonomy of such descriptive objects, as well as facility for defining new condition types.

14.1. Exceptions

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

Note: This specification follows SRFI 34 [29].

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

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

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

When a safe script 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 scripts are executed, but the intent is that users perceive the raising of an exception as a controlled escape from the situation that raised the exception, not as a crash.

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

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

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

Syntax: Each ⟨clause⟩ should have the same form as a `cond` clause. (Section 9.5.5.)

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

`(raise` *obj*`)` procedure

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

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

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

```
(call-with-current-continuation
 (lambda (k)
   (with-exception-handler
     (lambda (x)
       (display "condition: ")
       (write x)
       (newline)
       (k 'exception))
     (lambda ()
       (+ 1 (raise 'an-error))))))
```

prints: `condition: an-error`

⟹ `exception`

```
(call-with-current-continuation
 (lambda (k)
   (with-exception-handler
     (lambda (x)
       (display "something went wrong")
       (newline)
       'dont-care)
     (lambda ()
       (+ 1 (raise 'an-error))))))
```
prints: `something went wrong`
and then interrupts the program, reporting a `&non-continuable` *exception*

```
(guard (condition
         (else
          (display "condition: ")
          (write condition)
          (newline)
          'exception))
  (+ 1 (raise 'an-error)))
```
prints: `condition: an-error`
⟹ `exception`

```
(guard (condition
         (else
          (display "something went wrong")
          (newline)
          'dont-care))
 (+ 1 (raise 'an-error)))
```
prints: `something went wrong`
⟹ `dont-care`

```
(call-with-current-continuation
 (lambda (k)
   (with-exception-handler
     (lambda (x)
       (display "reraised ") (write x) (newline)
       (k 'zero))
     (lambda ()
       (guard (condition
                ((positive? condition) 'positive)
                ((negative? condition) 'negative))
         (raise 1))))))
```
⟹ `positive`

```
(call-with-current-continuation
 (lambda (k)
   (with-exception-handler
     (lambda (x)
       (display "reraised ") (write x) (newline)
       (k 'zero))
     (lambda ()
       (guard (condition
                ((positive? condition) 'positive)
                ((negative? condition) 'negative))
         (raise -1))))))
```
⟹ `negative`

```
(call-with-current-continuation
 (lambda (k)
   (with-exception-handler
     (lambda (x)
       (display "reraised ") (write x) (newline)
       (k 'zero))
     (lambda ()
       (guard (condition
                ((positive? condition) 'positive)
                ((negative? condition) 'negative))
         (raise 0))))))
```
prints: `reraised 0`
⟹ `zero`

```
(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'a 42))))
```
⟹ `42`

```
(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'b 23))))
```
⟹ `(b . 23)`

```
(with-exception-handler
  (lambda (x)
    42)
  (lambda ()
    (+ (raise-continuable #f)
       23)))
```
⟹ `65`

14.2. Conditions

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

Note: This specification follows SRFI 35 [30].

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

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

14.2.1. Condition objects

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

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

A condition belonging to several condition types with a common supertype may have distinct values for the supertype's fields for each type. The type used to access a field determines which of the values is returned. The program can extract each of these field values separately.

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

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

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

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

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

`(make-condition` *type field-name obj ...*`)` procedure

Returns a condition object belonging to condition type *type*. *Field-name* must be a field name. There must be a pair of a *field-name* and an *obj* for each field of *type* and its direct and indirect supertypes. The `make-condition` procedure returns the condition value, with the argument values associated with their respective fields.

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

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

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

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

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

Field-name must be a symbol. Moreover, *condition* must belong to a condition type which has a field name called *field-name*, or one of its (direct or indirect) supertypes must have the field. The `condition-ref` procedure returns the value associated with *field-name*.

`(make-compound-condition` $condition_1$ $condition_2$ `...)` procedure

Returns a compound condition belonging to all condition types that the *condition*s belong to.

The `condition-ref` procedure, when applied to a compound condition returns the value from the first of the *condition*s that has such a field.

`(extract-condition` *condition condition-type*`)` procedure

Condition must be a condition belonging to *condition-type*. The `extract-condition` procedure returns a condition of condition type *condition-type* with the field values specified by *condition*.

If *condition* is a compound condition, `extract-condition` extracts the field values from the subcondition belonging to *condition-type* that appeared first in the call to `make-compound-condition` that created the condition. The returned condition may be newly created; it is possible for

```
(let* ((&c (make-condition-type
             'c &condition '()))
       (c0 (make-condition &c))
       (c1 (make-compound-condition c0)))
  (eq? c0 (extract-condition c1 &c)))
```

to return `#f`.

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

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

`(`⟨field⟩ ⟨accessor⟩`)`

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

Semantics: The `define-condition-type` form defines an identifier ⟨condition-type⟩ to some value describing a new condition type. ⟨Supertype⟩ must be the name of a previously defined condition type.

The `define-condition-type` form also defines ⟨predicate⟩ to a predicate that identifies conditions associated with that type, or with any of its subtypes.

The `define-condition-type` form defines each ⟨accessor⟩ to a procedure which extracts the value of the named field from a condition associated with this condition type.

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

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

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

Each ⟨field-binding⟩ must be of the form

`(`⟨field⟩ ⟨expression⟩`)`

where ⟨field⟩ is a field identifier from the definition of ⟨condition-type⟩. x The ⟨expression⟩ are evaluated in some unspecified order; their values can later be extracted from the condition object via the accessors of the associated condition types or their supertypes.

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

```
(make-compound-condition
  (make-condition ⟨condition-type⟩ '⟨field-name⟩ ⟨value⟩ ...)
  ...)
```

with the condition types retaining their order from the `condition` form. The field names and values are duplicated as necessary as described below.

Each ⟨type-field-binding⟩ must contain field bindings for all fields of ⟨condition-type⟩ without duplicates. There is an exception to this rule: if a field binding is missing, and the field belongs to a supertype shared with one of the other ⟨type-field-binding⟩ subforms, then the value defaults to that of the first such binding in the condition form.

`&condition` condition type

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14.3. Standard condition types

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

This type describes conditions that can safely be ignored.

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

This type describes a violation of an implementation restriction allowed by the specification, such as the absence of representations for NaNs and infinities. (See section 16.4.)

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

This condition type could be defined by

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

This type describes defects in the program.

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

This type describes unbound identifiers in the program.

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

This condition type could be defined by

```
(define-condition-type &contract &defect
  contract-violation?)
```

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

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

This condition type could be defined by

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

The `irritants` field should contain a list of objects. This condition provides additional information about a condition, typically the argument list of a procedure that detected an exception. Conditions of this type are created by the `error` and `contract-violation` procedures of section 9.17.

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

This condition type could be defined by

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

The `who` field should contain a symbol or string identifying the entity reporting the exception. Conditions of this type are created by the `error` and `contract-violation` procedures (section 9.17), and the `syntax-violation` procedure (section 17.9).

15. I/O

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

- The `(r6rs i/o primitive)` library (section 15.2) is a simple, primitive I/O subsystem providing unbuffered I/O. Its primary purpose is to allow programs to implement custom data sources and sinks.
- The `(r6rs i/o ports)` library (section 15.3) is an I/O layer for conventional, imperative buffered input and output with mixed text and binary data.
- The `(r6rs i/o simple)` library (section 15.4) is a convenience library atop the `(r6rs i/o ports)` library for textual I/O, compatible with the traditional Scheme I/O procedures [28].

Section 15.1 defines a condition-type hierarchy common to the `(r6rs i/o primitive)` and `(r6rs i/o ports)` libraries.

15.1. Condition types

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

The condition types and corresponding predicates and accessors are exported by both the `(r6rs i/o primitive)` and `(r6rs i/o simple)` libraries.

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

This condition type could be defined by

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

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

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

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

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

This condition type could be defined by

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

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

15.2. Primitive I/O

This section defines the `(r6rs i/o primitive)` library, a simple, primitive I/O subsystem. It provides unbuffered I/O, and is close to what a typical operating system offers. Thus, its interface is suitable for implementing high-throughput and zero-copy I/O.

This library also allows programs to implement custom data sources and sinks via a simple interface. It handles only blocking-I/O.

15.2.1. 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 the implementation's underlying operating system. A *filename* parameter name means that the corresponding argument must be a file name.

Some implementations will provide a more abstract representation of file names. Indeed, most operating systems do not use strings for representing file names, but rather byte or word sequences. Moreover, the string notation is difficult to manipulate, and it is not portable across operating systems.

15.2.2. 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 19) over the symbols constituting valid file options. A *file-options* parameter name means that the corresponding argument must be a file-options object.

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

Each ⟨file-options name⟩ must be an ⟨identifier⟩. The `file-options` syntax returns a file-options object that encapsulates the specified options. The following options (all affecting output only) have predefined meaning:

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

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

The file-options object returned by `(file-options)` specifies, when supplied to an operation opening a file for output, that the file must exist (otherwise an exception with condition type `&i/o-file-exists-not` is raised) and its data is unchanged by the operation.

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

15.2.3. Readers and writers

The objects representing input data sources are called readers, and those representing output data sinks are called writers. Both readers and writer are unbuffered, and they operate purely on binary data. Although some reader and writer objects might conceivably have something to do with files or devices, programmers should never assume it. A *reader* parameter name means that the corresponding argument must be a reader. A *writer* parameter name means that the corresponding argument must be a writer.

The `(r6rs i/o primitive)` library has one condition type specific to readers and writers:

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

This condition type could be defined by

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

This condition type allows specifying the particular reader or writer with which an I/O error is associated. Conditions raised by the procedures exported by the `(r6rs i/o primitive)` may include an `&i/o-reader/writer` condition, but they are not required to do so.

15.2.4. I/O buffers

`(make-i/o-buffer` *size*`)` procedure

Size must be a non-negative exact integer. The `make-i/o-buffer` procedure creates a bytes object of size *size* with undefined contents. Callers of the procedures from the `(r6rs i/o primitive)` library are encouraged to use bytes objects created by `make-i/o-buffer`, because they might have alignment and placement characteristics that `make reader-read!` and `writer-write!` more efficient. (Those procedures are still required to work on regular bytes objects, however.)

15.2.5. Readers

The purpose of reader objects is to represent the output of arbitrary algorithms in a form susceptible to imperative I/O.

`(reader?` *obj*`)` procedures

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

`(make-simple-reader` *id descriptor* procedure
chunk-size read! available get-position
set-position! end-position close`)`

Returns a reader object. *Id* must be a string naming the reader, provided for informational purposes only. *Descriptor* may be any object; the procedures described in this section do not use it internally for any purpose, but *descriptor* can be extracted from the reader object via `reader-descriptor`. Thus, *descriptor* can be used to keep the internal state of certain kinds of readers.

Chunk-size must be a positive exact integer, and it represents a recommended efficient size for the read operations on this reader. This integer is typically the block size of the buffers of the operating system. As such, it is only a hint for clients of the reader—calls to the *read!* procedure (see below) may specify a different read count.

The remaining arguments are procedures. For each such procedure, an operation exists that calls that procedure. For example, the `reader-read!` operation, when called on a simple reader, calls its *read!* procedure. It is encouraged that these procedures check that their arguments are of the appropriate types. The operations that call them perform no checking beyond ensuring that their *reader* arguments are simple readers, and, if applicable, that the reader supports the operation. These procedures must raise exceptions with condition types as specified above when they encounter an exceptional situation. When they do not, the effects are unspecified.

Get-position, *set-position!*, and *end-position* may be omitted, in which case the corresponding arguments must be `#f`.

- (*read! bytes start count*)

  Start and *count* must be non-negative exact integers. The *read!* procedure reads up to *count* bytes from the reader and writes them into *bytes* starting at index start. *Bytes* must have size at least *start* + *count*. The result is the number of bytes read as an exact integer. The result is 0 if *read!* encounters an end of file, or if count is 0. If *count* is positive, *read!* blocks until at least one byte has been read or until it has encountered an end of file.

Bytes may or may not be a bytes object returned by `make-i/o-buffer`.

Count may or may not be the same as the chunk size of the reader.

- (*available*)

  Returns an estimate of the total number of bytes left in the reader. The result is either an exact integer, or `#f` if no such estimate is possible. There is no guarantee that this estimate has any specific relationship to the true number of available bytes.

- (*get-position*)

  When present, *get-position* returns the current position in the reader as an exact integer, counting the number of bytes since the beginning of the source. (Ends of file do not count as bytes.)

- (*set-position! pos*)

  When present, *set-position!* moves to position *pos* (which must be a non-negative exact integer) in the reader.

- (*end-position*)

  When present, *end-position* returns the position in the reader of the next end of file, without changing the current position.

- (*close*)

  Marks the reader as closed, performs any necessary cleanup, and releases the resources associated with the reader. Further operations on the reader must raise an exception with condition type `&contract`.

`(reader-id` *reader*`)`

Returns a string naming the reader, provided for informational purposes only. For a file reader returned by `open-file-reader` or `open-file-reader+writer`, the result is a string representation of the file name.

For a reader created by `make-simple-reader`, the result is the value that was supplied as the *id* argument to `make-simple-reader`.

`(reader-descriptor` *reader*`)` procedure

For a reader created by `make-simple-reader`, the result is the value that was supplied as the descriptor argument to `make-simple-reader`.

For all other readers, the result is an unspecified value.

`(reader-chunk-size` *reader*`)` procedure

Returns a positive exact integer that represents a recommended efficient size of the read operations on this reader. The result is typically the block size of the buffers of the operating system. As such, it is only a hint for clients of the reader—calls to the `reader-read!` procedure (see below) may specify a different read count.

For a reader created by `make-simple-reader`, the result is the value that was supplied as the chunk-size argument to `make-simple-reader`.

`(reader-read!` *reader bytes start count*`)` procedure

Start and *count* must be non-negative exact integers. *Bytes* must have at least *start* + *count* elements. The `reader-read!` procedure reads up to *count* bytes from the reader and writes them into *bytes* starting at index *start*. The result is the number of bytes read as an exact integer. The result is 0 if `reader-read!` encounters an end of file, or if *count* is 0. If *count* is positive, `reader-read!` blocks until at least one byte has been read or until it has encountered end of file.

Bytes may or may not be a bytes object returned by `make-i/o-buffer`, but `reader-read!` may operate more efficiently if it is.

Count may or may not be the same as the chunk size of the reader, but `reader-read!` may operate more efficiently if it is.

For a reader created by `make-simple-reader`, `reader-read!` tail-calls the *read!* procedure of reader with the remaining arguments.

`(reader-available` *reader*`)` procedure

Returns an estimate of the total number of available bytes left in the stream. The result is either an exact integer, or `#f` if no such estimate is possible. There is no guarantee that this estimate has any specific relationship to the true number of available bytes.

For a reader created by `make-simple-reader`, `reader-available` tail-calls the *available* procedure of reader.

`(reader-has-get-position?` *reader*`)` procedure
`(reader-get-position` *reader*`)` procedure

The `reader-has-get-position?` procedure `#t` if *reader* supports the `reader-get-position` procedure, and `#f` otherwise. For a simple reader, `reader-has-get-position?` returns `#t` if it has a *get-position* procedure.

When `reader-has-get-position` returns `#t` for reader, `reader-get-position` returns the current position in the

byte stream as an exact integer counting the number of bytes since the beginning of the stream. Otherweise, an exception with condition type `&contract` is raised.

For a reader created by `make-simple-reader`, `reader-get-position` tail-calls the *get-position* procedure of *reader*.

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

Pos must be a non-negative exact integer.

The `reader-has-set-position!?` procedure `#t` if *reader* supports the `reader-set-position!` procedure, and `#f` otherwise. For a simple reader, `reader-has-set-position!?` returns `#t` if it has a *set-position!* procedure.

When `reader-has-set-position!?` returns `#t` for *reader*, `reader-set-position!` moves to position *pos* in the stream. Otherwise, an `exception` with condition type `&contract` is raised.

For a reader created by `make-simple-reader`, `reader-set-position!` tail-calls the *set-position!* procedure of *reader* with the *pos* argument. For readers not created by `make-simple-reader`, the result values are unspecified.

`(reader-has-end-position?` *reader*`)` procedure
`(reader-end-position` *reader*`)` procedure

The `reader-has-end-position?` procedure `#t` if *reader* supports the `reader-end-position` procedure, and `#f` otherwise. For a simple reader, `reader-has-end-position?` returns `#t` if it has a *end-position* procedure.

When `reader-has-end-position?` returns `#t` for *reader*, `reader-end-position` returns the position in the byte stream of the next end of file, without changing the current position. Otherwise, an exception with condition type `&contract` is raised.

For a reader created by `mmake-simple-reader`, `reader-end-position` tail-calls the *end-position* procedure of *reader*.

`(reader-close` *reader*`)` procedure

Marks *reader* as closed, performs any necessary cleanup, and releases the resources associated with the reader. Further operations on the reader must raise an exception with condition type `&contract`.

For a reader created by `make-simple-reader`, `reader-close` tail-calls the *close* procedure of reader. For all other readers, the return values are unspecified.

`(open-bytes-reader` *bytes*`)` procedure

Returns a *bytes reader* that uses *bytes*, a bytes object, as its contents. The result reader supports the `reader-get-position`, `reader-set-position!`, and `reader-end-position` operations. The effect of modifying the contents of *bytes*, after `open-bytes-reader` has been called, on the reader is unspecified.

`(open-file-reader` *filename*`)` procedure
`(open-file-reader` *filename file-options*`)` procedure

Returns a reader connected to the file named by *filename*. The *file-options* object, which may determine various aspects of the returned reader (see section 15.2.2) defaults to `(file-options)` if not present. The result reader supports `reader-get-position`, `reader-set-position!`, and `reader-end-position` operations.

`(standard-input-reader)` procedure

Returns a reader connected to the standard input. The meaning of "standard input" is implementation-dependent.

15.2.6. Writers

The purpose of writer objects is to represent the input of arbitrary algorithms in a form susceptible to imperative I/O.

`(writer?` *obj*`)` procedure

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

`(make-simple-writer` *id descriptor* procedure
chunk-size write! get-position set-position!
end-position close`)`

Returns a writer object. *Id* must be a string naming the writer, provided for informational purposes only. *Descriptor* may be any object; the procedures described in this section do not use it internally for any purpose, but *descriptor* can be extracted from the writer object via `writer-descriptor`. Thus, *descriptor* can be used to keep the internal state of certain kinds of writers.

Chunk-size must be a positive exact integer, and it is the recommended efficient size of the write operations on this writer. As such, it is only a hint for clients of the writer—calls to the *write!* procedure (see below) may specify a different write count.

The remaining arguments are procedures. For each such procedure, an operation exists that calls that procedure. For example, the `writer-write!` operation, when called on a simple writer, calls its *write!* procedure. It is encouraged

that these procedures check that their arguments are of the appropriate types. The operations that call them perform no checking beyond ensuring that their *writer* arguments are indeed writers, and, if applicable, that the writer supports the operation. These procedures must raise exceptions with condition types as specified above when they encounter an exceptional situation. When they do not, the effects are unspecified.

Get-position, *set-position!*, and *end-position* may be omitted, in which case the corresponding arguments must be `#f`.

- (*write! bytes start count*)

  Start and *count* must be non-negative exact integers. The *write!* procedure writes up to *count* bytes in bytes object *bytes* starting at index *start*. Before writing any bytes, `write!` blocks until it can write at least one byte. The result is the number of bytes actually written as a positive exact integer.

  Bytes may or may not be a bytes object returned by `make-i/o-buffer`.

  Count may or may not be the same as the chunk size of the reader.

- (*get-position*)

  When present, *get-position* returns the current position in the byte stream as an exact integer counting the number of bytes since the beginning of the stream.

- (*set-position! pos*)

  When present, *set-position!* moves to position *pos* (which must be a non-negative exact integer) in the stream.

- (*end-position*)

  When present, *end-position* returns the byte position of the next end of file without changing the current position.

- (*close*)

  Marks the writer as closed, performs any necessary cleanup, and releases the resources associated with the writer. Further operations on the writer must raise an exception with condition type `&contract`.

`(writer-id` *writer*`)` procedure

Returns string naming the writer, provided for informational purposes only. For a file writer returned by `open-file-writer` or `open-file-reader+writer`, the result is a string representation of the file name.

For a writer created by `make-simple-writer`, the result is the value of the *id* field of the argument writer.

`(writer-descriptor` *writer*`)` procedure

For a writer created by `make-simple-writer`, `write-descriptor` returns the value of the *descriptor* field of the argument writer.

For all other writers, the result is an unspecified value.

`(writer-chunk-size` *writer*`)` procedure

Returns a positive exact integer, and is the recommended efficient size of the write operations on this writer. As such, it is only a hint for clients of the writer—calls to `writer-write!` (see below) may specify a different write count.

For a writer created by `make-simple-writer`, the result is the value of the *chunk-size* field of the argument writer.

`(writer-write!` *writer bytes start count*`)` procedure

Start and *count* must be non-negative exact integers. *Bytes* must have at least *start* + *count* elements. The `writer-write!` procedure writes up to *count* bytes in bytes object *bytes* starting at index *start*. Before writing any bytes, `write-write!` blocks until it can write at least one byte. The result is the number of bytes actually written as a positive exact integer.

Bytes may or may not be a bytes object returned by `make-i/o-buffer`, but `writer-write!` may operate more efficiently if it is.

Count may or may not be the same as the chunk size of the reader, but `writer-write!` may operate more efficiently if it is.

For a writer created by `make-simple-writer`, writer-write! tail-calls the *write!* procedure of writer with the remaining arguments. For all other writers, the result values are unspecified.

`(writer-has-get-position?` *writer*`)` procedure
`(writer-get-position` *writer*`)` procedure

The `writer-has-get-position?` procedure `#t` if *writer* supports the `writer-get-position` procedure, and `#f` otherwise. For a simple writer, `writer-has-get-position?` returns `#t` if it has a *get-position* procedure.

When `writer-has-get-position?` returns `#t` for *writer*, `writer-get-position` returns the current position in the byte stream as an exact integer counting the number of bytes since the beginning of the stream. Otherwise, an exception with condition type `&contract` is raised.

For a writer created by `make-simple-writer`, `writer-get-position` calls the *get-position* procedure of *writer*.

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

Pos must be a non-negative exact integer.

The `writer-has-set-position!?` procedure `#t` if *writer* supports the `writer-set-position!` procedure, and `#f` otherwise. For a simple writer, `writer-has-set-position!?` returns `#t` if it has a *set-position!* procedure.

When `writer-has-set-position!?` returns `#t` for *writer*, `writer-set-position!` moves to position *pos* (which must be a non-negative exact integer) in the stream. Otherwise, an exception with condition type `&contract` is raised.

For a writer created by `make-simple-writer`, `writer-set-position!` calls the *set-position!* procedure of *writer* with the *pos* argument. For writers not created by `make-simple-reader`, the result values are unspecified.

`(writer-has-end-position?` *writer*`)` procedure
`(writer-end-position` *writer*`)` procedure

The `writer-has-end-position?` procedure `#t` if *writer* supports the `writer-end-position` procedure, and `#f` otherwise. For a simple writer, `writer-has-end-position?` returns `#t` if it has a *end-position* procedure.

When `writer-has-end-position?` returns `#t` for writer, `writer-end-position` returns the byte position of the next end of file, without changing the current position. Otherwise, an exception with condition type `&contract` is raised.

For a writer created by `make-simple-writer`, `writer-end-position` calls the *end-position* procedure of *writer*.

`(writer-close` *writer*`)` procedure

Marks the writer as closed, performs any necessary cleanup, and releases the resources associated with the writer. Further operations on the writer must raise an exception with condition type `&contract`.

For a writer created by `make-simple-writer`, calls the *close* procedure of *writer*. For all other writers, the result values are unspecified.

`(open-bytes-writer)` procedure

Returns a *bytes writer* that can yield everything written to it as a bytes object. The result writer supports the `writer-get-position`, `writer-set-position!`, and `writer-end-position` operations. operations.

`(writer-bytes` *writer*`)` procedure

The *writer* argument must be a bytes writer. The `writer-bytes` procedure returns a bytes object containing the data written to writer in sequence. Doing this in no way invalidates the writer or change its store. The effect of modifying the contents of the returned bytes object on *writer* is unspecified.

`(clear-writer-bytes!` *writer*`)` procedure

Writer must be a bytes writer. The `clear-writer-bytes!` procedure clears the bytes object associated with *writer*, associating it with an empty bytes object.

`(open-file-writer` *filename*`)` procedure
`(open-file-writer` *filename file-options*`)` procedure

Returns a writer connected to the file named by *filename*. The *file-options* object, which determines various aspects of the returned writer (see section 15.2.2) defaults to `(file-options)` if not present. The result writer supports the `writer-get-position`, `writer-set-position!`, and `writer-end-position` operations.

`(standard-output-writer)` procedure

Returns a writer connected to the standard output. The meaning of "standard output" is implementation-dependent.

`(standard-error-writer)` procedure

Returns a writer connected to the standard error. The meaning of "standard error" is implementation-dependent.

15.2.7. Opening files for reading and writing

`(open-file-reader+writer` *filename*`)` procedure
`(open-file-reader+writer` *filename file-options*`)` procedure

Returns two values: a reader and a writer connected to the file named by *filename*. The *file-options* object, which determines various aspects of the returned writer and possibly the reader (see section 15.2.2), defaults to `(file-options)` if not present. The result reader supports the `reader-get-position`, `reader-set-position!`, `reader-end-position`, and the result writer supports the `writer-get-position`, `writer-set-position!`, `writer-end-position` operations.

Note: The `open-file-reader+writer` procedure enables opening a file for simultaneous input and output in environments where it is not possible to call `open-file-reader` and `open-file-writer` on the same file.

15.2.8. Examples

```
; Algorithmic reader producing an infinite
; stream of blanks:

(define (make-infinite-blanks-reader)
  (make-simple-reader
    "<blanks, blanks, and more blanks>"
    #f
    4096
    (lambda (bytes start count)
      (let loop ((index 0))
        (if (>= index count)
            index
            (begin
              (bytes-u8-set! bytes (+ start index) 32)
              (loop (+ 1 index))))))
    (lambda ()
      1000) ; some number
    #f #f #f
    unspecific))

; Sample implementation of bytes writer

(define-record-type (buffer
                     make-buffer buffer?)
  (fields (mutable bytes
                   buffer-bytes set-buffer-bytes!)
          (mutable size
                   buffer-set-size set-buffer-size!)))

(define (open-bytes-writer)
  (let ((buffer
         (make-buffer (make-bytes 512) 0))
        (pos 0))

    (define (ensure-open)
      (if (not buffer)
          (raise (condition
                  (&message
                   (message "bytes writer closed"))
                  (&contract)
                  (&i/o-reader/writer
                   (reader/writer writer))))))

    (define writer
      (make-writer
       "<bytes writer>"
       buffer
       3
       (lambda (bytes start count)
         (ensure-open)
         ;; resize buffer if necessary
         (let loop ((length
                     (bytes-length
                       (buffer-bytes buffer))))
           (cond
            ((> (+ pos count) length)
             (loop (* 2 length)))
            ((> length (bytes-length (car buffer)))
             (let ((new-buffer (make-bytes length)))
               (bytes-copy! (buffer-bytes buffer) 0
                            new-buffer 0
                            (buffer-size buffer))
               (set-buffer-bytes! buffer new-buffer)))))

         (bytes-copy! bytes start
                      (buffer-bytes buffer) pos
                      count)
         (set-buffer-size!
           buffer
           (max (buffer-size buffer) (+ pos count)))
         (set! pos (+ pos count))
         count)
       (lambda ()
         (ensure-open)
         pos)
       (lambda (new-pos)
         (ensure-open)
         (if (<= new-pos (buffer-size buffer))
             (set! pos new-pos)
             (raise
              (condition
               (&message
                (message "invalid position"))
               (&i/o-invalid-position
                (position new-pos))))))
       (lambda ()
         (ensure-open)
         (buffer-size buffer))
       (lambda ()
         (set-buffer-bytes! buffer #f))))
    writer))

(define (writer-bytes writer)
  (let* ((buffer (writer-descriptor writer))
         (target (make-bytes (buffer-size buffer))))
    (bytes-copy! (buffer-bytes buffer) 0
                 target 0
                 (buffer-size buffer))
    target))
```

15.3. Port I/O

The `(r6rs i/o ports)` library defines an I/O layer for conventional, imperative buffered input and output with mixed text and binary data. A *port* represents a buffered access object for a data sink or source or both simultaneously. The library allows creating ports from arbitrary input sources and sinks represented as readers and writers (see section 15.2), but does not require that all ports be built from readers and writers.

The `(r6rs i/o ports)` library distinguishes between *input ports* and *output ports*. An input port is a source for data, whereas an output port is a sink for data. A port may be both an input port and an output port; such a

port typically provides simultaneous read and write access to a file or other data.

This section uses the *input-port*, *output-port*, *port* parameter names for arguments that must be input ports (or combined input/output ports), output ports (or combined input/output ports), or any kind of port.

15.3.1. Condition type

This library introduces the following condition type:

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

This condition type could be defined by

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

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

15.3.2. Buffer modes

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

`(buffer-mode` ⟨name⟩`)` syntax

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

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

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

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

15.3.3. Text transcoders

Several different Unicode encoding schemes describe standard ways to encode characters and strings as byte sequences and to decode those sequences [51]. Within this document, a *codec* is a Scheme object that represents a Unicode or similar encoding scheme.

The text transcoders of this document generalize codecs to deal with common end-of-line conventions and different error-handling modes.

A *transcoder* is an opaque object that represents some specific bidirectional (but not necessarily lossless) translation between byte sequences and Unicode characters and strings. The transcoder specifies how procedures that perform textual input are to interpret input bytes as characters or strings. The transcoder also specifies how procedures that perform textual output are to translate characters into bytes. Moreover, the transcoder specifies a mode for handling encoding or decoding errors.

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

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

These are predefined codecs for the UTF-8, ISO8859-1, UTF-16LE, UTF-16BE, UTF32-LE, and UTF-32BE encoding schemes.

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

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

If *name* is one of the ⟨identifier⟩s `lf`, `cr`, `crlf`, or `ls`, then the form evaluates to the corresponding symbol. If *name* is not one of these identifiers, the effect of evaluating this expression is implementation-dependent.

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

`(native-eol-style)` procedure

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

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

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

An exception with this type is raised when one of the operations for textual output to a port encounters a character not supported by the codec of the specified transcoder. The `transcoder` field contains the transcoder specified with the operation.

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

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

This condition type could be defined by

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

An exception with this type is raised when one of the operations for textual output to a port encounters a character not supported by the codec of the specified transcoder. The `char` field of the condition object contains the character that the operation was unable to encode, and the `transcoder` field contains the transcoder specified with the operation.

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

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

If *name* is one of the ⟨identifier⟩s `ignore`, `raise`, or `replace`, then the result is the corresponding symbol. If *name* is not one of these identifiers, the result the expression is implementation-dependent.

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

When part of a transcoder, an error-handling mode specifies the behavior of a text I/O operation in the presence of an encoding or decoding error:

If a codec for an input port encounters an invalid or incomplete character encoding, it behaves according to the specified error-handling mode. If it is `ignore`, the first byte of the invalid encoding is ignored and decoding continues with the next byte. If it is `replace`, the replacement character U+FFFD is injected into the data stream. Decoding subsequently continues with the next byte. If it is `raise`, a continuable exception with condition type `&i/o-decoding` is raised. See the description of `&i/o-decoding` for details on how to handle such an exception.

If a codec for an output port encounters a character it cannot encode, it behaves according to the specified error-handling mode. If it is `ignore`, the character is ignored and encoding continues after the encoding. If it is `replace`, an encoding-specific replacement character is emitted by the transcoder, and decoding continues after the encoding. This replacement character is U+FFFD for the Unicode encodings capable of representing it, and the `?` character for the Latin-1 encoding. If the mode is `raise`, an exception with condition type `&i/o-encoding` is raised. See the description of `&i/o-decoding` for details on how to handle such an exception.

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

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

A transcoder returned by `make-transcoder` is equal in the sense of `eqv?` to any other transcoder returned by `make-transcoder`, if and only if the *code*, *eol-style*, and *handling-mode* arguments are equal in the sense of `eqv?`.

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

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

15.3.4. Input and output ports

The operations described in this section are common to input and output ports. A port may have an associated *position* that specifies a particular place within its

data sink or source as a byte count from the beginning of the sink or source, and operations for inspecting and setting it. (Ends of file do not count as bytes.) A port may also have an associated transcoder that represents a default text encoding associated with the port. Note that the transcoder associated with a port does not have any direct effect on the behavior of procedures that perform textual I/O except for `get-output-string` and `call-with-string-output-port`. However, it can be passed to those procedures to specify that the text encoding or decoding should happen according to that transcoder.

`(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 it has one, or `#f` if no transcoder is associated with *port*.

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

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

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

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

Pos must be a non-negative exact integer.

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

The `set-port-position!` procedure sets the current byte position of the port to *pos*. If *port* is an output or combined input and output port, this first flushes *port*. (See `flush-output-port`, section 15.3.6.) This procedure raises an exception with condition type `&contract` if the port does not support the operation.

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

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

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

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

15.3.5. Input ports

An input port allows reading an infinite sequence of bytes 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. All of the procedures that perform textual input accept a transcoder as an optional argument. If no transcoder is supplied or the *transcoder* argument is `#f`, the input bytes are interpreted as UTF-8 with a platform-specific end-of-line convention.

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 combined input and output port), and returns `#f` otherwise.

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

Returns `#t` if the `lookahead-u8` procedure would return the end-of-file object, and returns `#f` otherwise.

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

Returns an input port for the named file. The file-options object, which may determine various aspects of the returned port (see section 15.2.2), defaults to `(file-options)`.

The returned input port supports the `port-position` and `set-port-position!` operations.

If *transcoder* is specified, it becomes the transcoder associated with the returned port.

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

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

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

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

Returns an input port whose bytes are drawn from an encoding of *string*. If *transcoder* is not specified, that encoding is UTF-8. If *transcoder* is specified, the encoding is according to *transcoder*. Also, *transcoder* becomes the transcoder associated with the returned port. The effect of modifying *string*, after `open-string-input-port` has been called, on the returned port is unspecified.

`(standard-input-port)` procedure

Returns an input port connected to standard input, possibly a fresh one on each call. If the returned port supports textual input, it has an associated transcoder with some encoding, some end-of-line style, and error-handling mode `raise`.

Note: Implementations are encouraged to provide a transcoder appopriate for reading text from the port.

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

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

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

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

`(get-bytes-n` *input-port k*`)` procedure

Reads from *input-port*, blocking as necessary, until k bytes are available from *input-port* or until an end of file is reached. If k or more bytes are available before an end of file, `get-bytes-n` returns a bytes object of size k. If fewer bytes are available before an end of file, `get-bytes-n` returns a bytes object 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-bytes-n` returns the end-of-file object.

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

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

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

`(get-bytes-some` *input-port*`)` procedure

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

`(get-bytes-all` *input-port*`)` procedure

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Any ⟨intertoken space⟩ (see section 3.2) in the input is first skipped. If an end of file occurs after the ⟨intertoken space⟩, the end of file object (see section 9.9) 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 not parsable, an exception with condition types `&lexical` and `&i/o-read` is raised.

15.3.6. Output ports

An output port is a sink to which bytes are written. These bytes may control external devices, or may produce files and other objects that may subsequently be opened for input. The procedures in this section that perform textual output accept a transcoder as an optional argument. If no transcoder is supplied, the character(s) output is translated to UTF-8 with a platform-specific end-of-line convention.

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

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

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

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

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

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

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

Returns an output port for the named file and the specified options (which default to `(file-options)`). The *buffer-mode* argument is optional; it defaults to `block`. If *transcoder* is specified, it becomes the transcoder associated with the returned port.

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

Creates an output port that accumulates a bytes object from the output written to it. If *transcoder* is specified, it becomes the transcoder associated with the returned port.

The `open-string-output-port` procedure is the same as `open-bytes-output-port`.

`(get-output-bytes` *output-port*`)` procedure

Output-port must be a port that accumulates a bytes object from the output written to it, such as the ports created by `open-bytes-output-port` and `call-with-bytes-output-port`. Returns a bytes object containing the output that has been accumulated with *output-port* so far. If the returned bytes object is modified, the the effect on *output-port* is unspecified.

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

Proc must be a procedure accepting one argument. Creates an output port that accumulates a bytes object from the output written to it, and calls *proc* with that output port as an argument. When *proc* returns for the first time, the port is closed and the bytes object associated with the port is returned. If *transcoder* is specified, it becomes the transcoder associated with the port.

`(get-output-string` *output-port*`)` procedure
`(get-output-string` *output-port transcoder*`)` procedure

Output-port must be a port that accumulates a bytes object from the output written to it, such as the ports created by `open-bytes-output-port` and `call-with-bytes-output-port`. The decoding of the bytes object associated with *output-port* is returned as a string. That decoding is according to *transcoder* if it is specified. If it is not specified, the decoding is according to the transcoder associated with the port. If no transcoder is associated with the port, the decoding is according to UTF-8. In either case, decoding errors are always handled analogously to the `replace` error-handling mode: the first byte of each invalid encoding is skipped and decoded as the U+FFFD replacement character.

If the returned string is modified, the effect on *output-port* is unspecified.

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

Proc must be a procedure accepting one argument. Creates an output port that accumulates a bytes object from the output written to it, and calls *proc* with that port as an argument. When *proc* returns for the first time, the port is closed and the decoding of the bytes object associated with the port is returned as a string. That decoding is according to *transcoder* if it is specified. If it is not specified, the decoding is according to the transcoder associated with the port. If no transcoder is associated with the port, the decoding is according to UTF-8. In either case, decoding errors are always handled analogously to the `replace` error-handling mode. If *transcoder* is specified, it also becomes the transcoder associated with the port.

`(clear-bytes-output-port!` *output-port*`)` procedure
`(clear-string-output-port!` *output-port*`)` procedure

Output-port must be a port that accumulates a bytes object from the output written to it, such as the ports created by the `open-bytes-output-port` and `call-with-bytes-output-port` procedures. The `clear-bytes-output-port!` and `clear-string-output-port!` procedures clear the bytes object associated with *output-port*, associating it with an empty bytes object.

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

Returns a port connected to the standard output or standard error, respectively. If the returned port supports textual input, it has an associated transcoder with some encoding, some end-of-line style, and error-handling mode `raise`.

Note: Implementations are encouraged to provide transcoders appopriate for writing text to these ports.

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

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

`(put-bytes` *output-port bytes*`)` procedure
`(put-bytes` *output-port bytes start*`)` procedure
`(put-bytes` *output-port bytes start count*`)` procedure

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

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

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

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

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

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

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

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

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

Note: The `put-datum` procedure merely writes the external reprentation. If `put-datum` is used to write several subsequent external representations to an output port, care must be taken to delimit them properly so they can be read back in by subsequent calls to `get-datum`.

15.3.7. Opening files for reading and writing

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

Returns a single port that is both an input port and an output port for the named file and options (which default to `(file-options)`). *buffer-mode* optionally specifies the buffer mode of the port; it defaults to `block`.

The returned port supports the `port-position` and `set-port-position!` operations. The same port position is used for both input and output.

If *transcoder* is specified, it becomes the transcoder associated with the returned port.

15.3.8. Ports from readers and writers

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

Returns an input port connected to the reader *reader*. If *transcoder* is specified, it becomes the transcoder associated with the returned port.

The returned port supports the `port-position` operation if and only if *reader* supports the `reader-get-position` operation. It supports the `set-port-position!` operation if and only if *reader* supports the `reader-set-position!` operation.

`(open-writer-output-port` *writer*`)` procedure
`(open-writer-output-port` *writer buffer-mode*`)` procedure
`(open-writer-output-port` *writer buffer-mode transcoder*`)` procedure

Returns an output port connected to the writer *writer*. *buffer-mode* optionally specifies the buffer mode of the port; it defaults to `block`. If *transcoder* is specified, it becomes the transcoder associated with the returned port.

The returned port supports the `port-position` operation if and only if *writer* supports the `writer-get-position` operation. It supports the `set-port-position!` operation if and only if *writer* supports the `writer-set-position!` operation.

`(open-reader/writer-input/output-port` *reader writer buffer-mode transcoder*`)` procedure

Returns a combined input/output port connected to *reader* for reading operations and to *writer* for writing operations. *Buffer-mode* optionally specifies the buffer mode of the port; it defaults to `block`. If *transcoder* is specified, it becomes the transcoder associated with the returned port.

The returned port supports the `port-position` operation if and only if *reader* supports the `reader-get-position` operation and *writer* supports the `writer-get-position` operation. It supports the `set-port-position!` operation if and only if *reader* supports the `reader-set-position!` operation and *writer* supports the `writer-set-position!` operation. In that case, each call to `set-port-position!` will set the position of both *reader* and *writer*.

Note: It is expected that *reader* and *writer* access the same underlying data, as is the case with the reader and writer and writer returned by `open-file-reader+writer`.

15.4. Simple I/O

This section describes the `(r6rs i/o simple)` library, which provides a somewhat more convenient interface for performing textual I/O on ports. This library implements most of the I/O procedures of the previous version of this report [28].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This reads from *input-port* using the transcoder assocated with it, blocking as necessary, until the complete encoding for a character is available from input-port, or the bytes that are available cannot be the prefix of any valid encoding, or an end of file is reached.

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

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

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

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

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

Reads an external representation from *input-port* using the transcoder associated with *input-port* and returns the datum it represents. The `read` procedure operates in the same way as `get-datum`, see section 15.3.5.

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

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

Writes an encoding of the character *char* to the port using the transcoder associated with *output-port*. The unspecified value is returned.

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

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

This is equivalent to using `write-char` to write `#\linefeed` to *output-port* using the transcoder associated with *output-port*.

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

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

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

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

Writes the external representation of *obj* to *output-port* using the transcoder associated with *output-port*. The `write` procedure operates in the same way as `put-datum`, see section 15.3.6.

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

16. Arithmetic

This chapter describes Scheme's libraries for more specialized numerical operations: fixnum and flonum arithmetic, as well as generic exact and generic inexact arithmetic. It also gives more precise descriptions of the semantics of infinities and NaNs, and some of the underlying mathematical operations for integer division and the transcendental functions.

16.1. Representability of infinities and NaNs

The specification of the numerical operations is written as though infinities and NaNs are representable, and specifies many operations with respect to these numbers in ways that are consistent with the IEEE 754 standard for binary floating point arithmetic. An implementation of Scheme is not required to represent infinities and NaNs, however; an implementation must raise a continuable exception with condition type `&no-infinities` or `&no-nans` (respectively; see section 16.4) whenever it is unable to represent an infinity or NaN as required by the specification. In this case, the continuation of the exception handler is the continuation that otherwise would have received the infinity or NaN value. This requirement also applies to conversions between numbers and external representations, including the reading of program source code.

16.2. Semantics of common operations

Some operations are the semantic basis for several arithmetic procedures. The behavior of these operations is described in this section for later reference.

16.2.1. Integer division

For various kinds of arithmetic (fixnum, flonum, exact, inexact, and generic), Scheme provides operations for performing integer division. They rely on mathematical operations div, mod, div_0, and mod_0, that are defined as follows:

div, mod, div_0, and mod_0 each accept two real numbers x_1 and x_2 as operands, where x_2 must be nonzero.

div returns an integer, and mod returns a real. Their results are specified by

$$\begin{aligned} x_1 \text{ div } x_2 &= n_d \\ x_1 \text{ mod } x_2 &= x_m \end{aligned}$$

where

$$\begin{gathered} x_1 = n_d * x_2 + x_m \\ 0 \le x_m < |x_2| \end{gathered}$$

Examples:

$$5 \text{ div } 3 = 1$$

$$\begin{aligned} 5 \text{ div } -3 &= -1 \\ 5 \text{ mod } 3 &= 2 \\ 5 \text{ mod } -3 &= 2 \end{aligned}$$

div_0 and mod_0 are like div and mod, except the result of mod_0 lies within a half-open interval centered on zero. The results are specified by

$$\begin{aligned} x_1 \text{ div}_0 \ x_2 &= n_d \\ x_1 \text{ mod}_0 \ x_2 &= x_m \end{aligned}$$

where:

$$\begin{aligned} x_1 &= n_d * x_2 + x_m \\ -|\tfrac{x_2}{2}| &\le x_m < |\tfrac{x_2}{2}| \end{aligned}$$

Examples:

$$\begin{aligned} 5 \text{ div}_0 \ 3 &= 2 \\ 5 \text{ div}_0 \ -3 &= -2 \\ 5 \text{ mod}_0 \ 3 &= -1 \\ 5 \text{ mod}_0 \ -3 &= -1 \end{aligned}$$

Rationale: The half-open symmetry about zero is convenient for some purposes.

16.2.2. Transcendental functions

In general, the transcendental functions log, $\sin^{-1}$ (arcsine), $\cos^{-1}$ (arccosine), and $\tan^{-1}$ are multiply defined. The value of $\log z$ is defined to be the one whose imaginary part lies in the range from $-\pi$ (inclusive if -0.0 is distinguished, exclusive otherwise) to π (inclusive). $\log 0$ is undefined.

The value of $\log z$ for non-real z is defined in terms of log on real numbers as

$$\log z = \log |z| + \text{angle } z$$

where angle z is the angle of $z = a \cdot e^{ib}$ specified as:

$$\text{angle } z = b + 2\pi n$$

with $-\pi \le \text{angle } z \le \pi$ and angle $z = b + 2\pi n$ for some integer n.

With the one-argument version of log defined this way, the values of the two-argument-version of log, $\sin^{-1} z$, $\cos^{-1} z$, $\tan^{-1} z$, and the two-argument version of $\tan^{-1}$ are according to the following formulæ:

$$\begin{aligned} \log z \ b &= \frac{\log z}{\log b} \\ \sin^{-1} z &= -i \log(iz + \sqrt{1 - z^2}) \\ \cos^{-1} z &= \pi/2 - \sin^{-1} z \\ \tan^{-1} z &= (\log(1 + iz) - \log(1 - iz))/(2i) \\ \tan^{-1} x \ y &= \text{angle}(x + yi) \end{aligned}$$

The range of $\tan^{-1} x \ y$ is as in the following table. The asterisk (*) indicates that the entry applies to implementations that distinguish minus zero.

	y condition	x condition	range of result r
	$y = 0.0$	$x > 0.0$	0.0
*	$y = +0.0$	$x > 0.0$	$+0.0$
*	$y = -0.0$	$x > 0.0$	-0.0
	$y > 0.0$	$x > 0.0$	$0.0 < r < \frac{\pi}{2}$
	$y > 0.0$	$x = 0.0$	$\frac{\pi}{2}$
	$y > 0.0$	$x < 0.0$	$\frac{\pi}{2} < r < \pi$
	$y = 0.0$	$x < 0$	π
*	$y = +0.0$	$x < 0.0$	π
*	$y = -0.0$	$x < 0.0$	$-\pi$
	$y < 0.0$	$x < 0.0$	$-\pi < r < -\frac{\pi}{2}$
	$y < 0.0$	$x = 0.0$	$-\frac{\pi}{2}$
	$y < 0.0$	$x > 0.0$	$-\frac{\pi}{2} < r < 0.0$
	$y = 0.0$	$x = 0.0$	undefined
*	$y = +0.0$	$x = +0.0$	$+0.0$
*	$y = -0.0$	$x = +0.0$	-0.0
*	$y = +0.0$	$x = -0.0$	π
*	$y = -0.0$	$x = -0.0$	$-\pi$
*	$y = +0.0$	$x = 0$	$\frac{\pi}{2}$
*	$y = -0.0$	$x = 0$	$-\frac{\pi}{2}$

The above specification follows Steele [46], which in turn cites Penfield [38]; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions.

16.3. Fixnums

Every implementation must define its fixnum range as a closed interval

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

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

This section specifies two kinds of operations on fixnums. Operations whose names begin with `fixnum` perform arithmetic modulo 2^w. Operations whose names begin with `fx` perform integer arithmetic on their fixnum arguments, but raise an exception with condition type `&implementation-restriction` if the result is not a fixnum.

Rationale: The operations whose names begin with `fixnum` implement arithmetic on a quotient ring of the integers, but their results are not the same in every implementation because the particular ring is parameterized by w. The operations whose names begin with `fx` do not have as nice a closure property, and the arguments that cause them to raise an exception are not the same in every implementation, but any results they

return without raising an exception are the same in all implementations.

Some operations (e.g. `fixnum<` and `fx<`) behave the same in both sets.

Rationale: Duplication of names reduces bias toward either set, and saves programmers from having to remember which names are supplied.

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

16.3.1. Quotient-ring fixnum operations

This section describes the `(r6rs arithmetic fixnum)` library.

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

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

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

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

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

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

`(fixnum-zero?` *fx*`)` procedure
`(fixnum-positive?` *fx*`)` procedure
`(fixnum-negative?` *fx*`)` procedure
`(fixnum-odd?` *fx*`)` procedure
`(fixnum-even?` *fx*`)` procedure

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

`(fixnum-max` fx_1 fx_2 ...`)` procedure
`(fixnum-min` fx_1 fx_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

`(fixnum+` fx_1 ...`)` procedure
`(fixnum*` fx_1 ...`)` procedure

These procedures return the unique fixnum that is congruent mod 2^w to the sum or product of their arguments.

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

With two or more arguments, this procedure returns the unique fixnum that is congruent mod 2^w to the difference of its arguments, associating to the left. With one argument, however, it returns the the unique fixnum that is congruent mod 2^w to the additive inverse of its argument.

`(fixnum-div+mod` fx_1 fx_2`)` procedure
`(fixnum-div` fx_1 fx_2`)` procedure
`(fixnum-mod` fx_1 fx_2`)` procedure
`(fixnum-div0+mod0` fx_1 fx_2`)` procedure
`(fixnum-div0` fx_1 fx_2`)` procedure
`(fixnum-mod0` fx_1 fx_2`)` procedure

fx_2 must be nonzero. These procedures implement number-theoretic integer division modulo 2^w. Each procedure returns the unique fixnum(s) congruent modulo 2^w to the result(s) specified in section 16.2.1.

```
(fixnum-div ex1 ex2)          ⟹ ex1 div ex2
(fixnum-mod ex1 ex2)          ⟹ ex1 mod ex2
(fixnum-div+mod ex1 ex2)
        ⟹ ex1 div ex2, ex1 mod ex2
                              ; two return values
(fixnum-div0 ex1 ex2)         ⟹ ex1 div0 ex2
(fixnum-mod0 ex1 ex2)         ⟹ ex1 mod0 ex2
(fixnum-div0+mod0 ex1 ex2)
        ⟹ ex1 ex1 div0 ex2, ex1 mod0 ex2
                              ; two return values
```

`(fixnum+/carry` fx_1 fx_2 fx_3`)` procedure

Returns the two fixnum results of the following computation:

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

Note: The results returned by the `fixnum+/carry`, `fixnum-/carry`, and `fixnum*/carry` procedures depend upon the precision w, so there are no `fx` equivalents to these procedures.

`(fixnum-/carry` fx_1 fx_2 fx_3`)` procedure

Returns the two fixnum results of the following computation:

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

`(fixnum*/carry` fx_1 fx_2 fx_3`)` procedure

Returns the two fixnum results of the following computation:

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

`(fixnum-not` fx`)` procedure

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

`(fixnum-and` fx_1 ...`)` procedure
`(fixnum-ior` fx_1 ...`)` procedure
`(fixnum-xor` fx_1 ...`)` procedure

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

`(fixnum-if` fx_1 fx_2 fx_3`)` procedure

Returns the fixnum result of the following computation:

```
(fixnum-ior (fixnum-and fx1 fx2)
            (fixnum-and (fixnum-not fx1) fx3))
```

`(fixnum-bit-count` fx`)` procedure

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

`(fixnum-length` fx`)` procedure

Returns the fixnum result of the following computation:

```
(do ((result 0 (+ result 1))
     (bits (if (fixnum-negative? fx)
               (fixnum-not fx)
               fx)
           (fixnum-logical-shift-right bits 1)))
    ((fixnum-zero? bits)
     result))
```

`(fixnum-first-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.

```
(fixnum-first-bit-set 0)    ⟹  -1
(fixnum-first-bit-set 1)    ⟹  0
(fixnum-first-bit-set -4)   ⟹  2
```

`(fixnum-bit-set?` fx_1 fx_2`)` procedure

Fx_2 must be non-negative. The `fixnum-bit-set?` procedure returns the fixnum result of the following computation:

```
(not
  (fixnum-zero?
    (fixnum-and fx1
                (fixnum-logical-shift-left 1 fx2))))
```

`(fixnum-copy-bit` fx_1 fx_2 fx_3`)` procedure

Fx_2 must be non-negative. The `fixnum-copy-bit` procedure returns the fixnum result of the following computation:

```
(let* ((mask (fixnum-logical-shift-left 1 fx2)))
  (fixnum-if mask
             (fixnum-logical-shift-left fx3 fx2)
             fx1))
```

`(fixnum-bit-field` fx_1 fx_2 fx_3`)` procedure

Fx_2 and fx_3 must be non-negative. The `fixnum-bit-field` procedure returns the fixnum result of the following computation:

```
(let* ((mask (fixnum-not
               (fixnum-logical-shift-left -1 fx3))))
  (fixnum-logical-shift-right (fixnum-and fx1 mask)
                              fx2))
```

`(fixnum-copy-bit-field` fx_1 fx_2 fx_3 fx_4`)` procedure

Fx_2 and fx_3 must be non-negative. The `fixnum-copy-bit-field` procedure returns the fixnum result of the following computation:

```
(let* ((to    fx1)
       (start fx2)
       (end   fx3)
       (from  fx4)
       (mask1 (fixnum-logical-shift-left -1 start))
       (mask2 (fixnum-not
                (fixnum-logical-shift-left -1 end)))
       (mask (fixnum-and mask1 mask2)))
  (fixnum-if mask
             (fixnum-logical-shift-left from start)
             to))
```

`(fixnum-arithmetic-shift` *fx*$_1$ *fx*$_2$`)` procedure

Returns the unique fixnum that is congruent mod 2^w to the result of the following computation:

```
(exact-floor (* fx1 (expt 2 fx2)))
```

`(fixnum-arithmetic-shift-left` *fx*$_1$ *fx*$_2$`)` procedure
`(fixnum-arithmetic-shift-right` *fx*$_1$ *fx*$_2$`)` procedure

Fx$_2$ must be non-negative. `fixnum-arithmetic-shift-left` returns the same result as `fixnum-arithmetic-shift`, and `(fixnum-arithmetic-shift-right` *fx*$_1$ *fx*$_2$`)` returns the same result as `(fixnum-arithmetic-shift` *fx*$_1$ `(fixnum-` *fx*$_2$`))`.

`(fixnum-logical-shift-left` *fx*$_1$ *fx*$_2$`)` procedure

Behaves the same as `fixnum-arithmetic-shift-left`.

`(fixnum-logical-shift-right` *fx*$_1$ *fx*$_2$`)` procedure

Fx$_2$ must be non-negative. The `fixnum-logical-shift-right` procedure returns the result of the following computation:

```
(let* ((n       fx1)
       (shift   fx2)
       (shifted
         (fixnum-arithmetic-shift-right n shift)))
  (let* ((mask-width
           (fixnum-
             (fixnum-width)
             (fixnum-mod shift (fixnum-width))))
         (mask (fixnum-not
                 (fixnum-logical-shift-left
                   -1 mask-width))))
    (fixnum-and shifted mask)))
```

Note: The results of `fixnum-logical-shift-left` and `fixnum-logical-shift-left` can depend upon the precision w, so they have no `fx` equivalents.

`(fixnum-rotate-bit-field` *fx*$_1$ *fx*$_2$ *fx*$_3$ *fx*$_4$`)` procedure

Fx$_2$, *fx*$_3$, and *fx*$_4$ must be non-negative. The `fixnum-rotate-bit-field` procedure returns the result of the following computation:

```
(let* ((n     fx1)
       (start fx2)
       (end   fx3)
       (count fx4)
       (width (fixnum- end start)))
  (if (fixnum-positive? width)
      (let* ((count (fixnum-mod count width))
             (field0
               (fixnum-bit-field n start end))
             (field1
               (fixnum-logical-shift-left
                 field0 count))
             (field2
               (fixnum-logical-shift-right
                 field0 (fixnum- width count)))
             (field (fixnum-ior field1 field2)))
        (fixnum-copy-bit-field n start end field))
      n))
```

`(fixnum-reverse-bit-field` *fx*$_1$ *fx*$_2$ *fx*$_3$`)` procedure

Returns the fixnum obtained from *fx*$_1$ by reversing the bit field specified by *fx*$_2$ and *fx*$_3$.

```
(fixnum-reverse-bit-field #b1010010 1 4)
          ⟹  88 ; #b1011000
(fixnum-reverse-bit-field #b1010010 91 -4)
          ⟹  82 ; #b1010010
```

16.3.2. Signalling fixnum operations

This section describes the `(r6rs arithmetic fx)` library.

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

This is the same as `fixnum?` in the `(r6rs arithmetic fixnum)` library.

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

These are the same as `fixnum-width`, `least-fixnum`, and `greatest-fixnum` in the `(r6rs arithmetic fixnum)` library, respectively.

`(fx=?` *fx*$_1$ *fx*$_2$ *fx*$_3$ ...`)` procedure
`(fx>?` *fx*$_1$ *fx*$_2$ *fx*$_3$ ...`)` procedure
`(fx<?` *fx*$_1$ *fx*$_2$ *fx*$_3$ ...`)` procedure
`(fx>=?` *fx*$_1$ *fx*$_2$ *fx*$_3$ ...`)` procedure
`(fx<=?` *fx*$_1$ *fx*$_2$ *fx*$_3$ ...`)` procedure

These procedures perform the same operations as `fixnum=?`, `fixnum>?`, `fixnum<?`, `fixnum>=?`, and `fixnum<=?`, respectively.

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

These procedures perform the same operations as `fixnum-zero?`, `fixnum-positive?`, `fixnum-negative?`, `fixnum-odd?`, and `fixnum-even?`, respectively.

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

These procedures perform the same operations as `fixnum-max`, and `fixnum-min`, respectively.

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

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

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

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

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

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

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

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

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

Fx_2 must be nonzero. These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in the section 16.2.1. An exception with condition type `&implementation-restriction` is raised if a result specified by that section is not a fixnum.

```
(fxdiv ex1 ex2)              ⟹ ex1 div ex2
(fxmod ex1 ex2)              ⟹ ex1 mod ex2
(fxdiv+mod ex1 ex2)
          ⟹ ex1 div ex2, ex1 mod ex2
                             ; two return values
(fxdiv0 ex1 ex2)             ⟹ ex1 div0 ex2
(fxmod0 ex1 ex2)             ⟹ ex1 mod0 ex2
(fxdiv0+mod0 ex1 ex2)
          ⟹ ex1 ex1 div0 ex2, ex1 mod0 ex2
                             ; two return values
```

`(fxnot` fx`)` procedure

This performs the same operation as `fixnum-not`.

`(fxand` fx_1 ...`)` procedure
`(fxior` fx_1 ...`)` procedure
`(fxxor` fx_1 ...`)` procedure

These procedures perform the same operations as `fixnum-and`, `fixnum-ior`, and `fixnum-xor`, respectively.

`(fxif` fx_1 fx_2 fx_3`)` procedure

This performs the same operation as `fixnum-if`.

`(fxbit-count` fx`)` procedure

This performs the same operation as `fixnum-bit-count`.

`(fxlength` fx`)` procedure

This performs the same operation as `fixnum-length`.

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

This performs the same operation as `fixnum-first-bit-set`.

`(fxbit-set?` fx fx_2`)` procedure

Fx_2 must be non-negative and less than `(fixnum-width)`. The `fxbit-set?` procedure returns the same result as `fixnum-bit-set?`.

`(fxcopy-bit` fx_1 fx_2 fx_3`)` procedure

Fx_2 must be non-negative and less than `(fixnum-width)`. Fx_3 must be 0 or 1. The `fxcopy-bit` procedure returns the same result as `fixnum-copy-bit`.

`(fxbit-field` fx_1 fx_2 fx_3`)` procedure

Fx_2 and fx_3 must be non-negative and less than `(fixnum-width)`. Moreover, fx_2 must be less than or equal to fx_3. The `fxbit-field` procedure returns the same result returned by `fixnum-bit-field`.

`(fxcopy-bit-field` fx_1 fx_2 fx_3 fx_4`)` procedure

Fx_2 and fx_3 must be non-negative and less than `(fixnum-width)`. Moreover, or fx_2 must be less than or equal to fx_3. The `fxcopy-bit-field` procedure returns the same result returned by `fixnum-copy-bit-field`.

`(fxarithmetic-shift` fx_1 fx_2`)` procedure

The absolute value of the fx_2 must be less than `(fixnum-width)`. If

```
(exact-floor (* fx1 (expt 2 fx2)))
```

is a fixnum, then that fixnum is returned. Otherwise an exception with condition type `&implementation-restriction` is raised.

`(fxarithmetic-shift-left` fx_1 fx_2`)` procedure
`(fxarithmetic-shift-right` fx_1 fx_2`)` procedure

Fx_2 must be non-negative. `fxarithmetic-shift-left` behaves the same as `fxarithmetic-shift`, and `(fxarithmetic-shift-right` fx_1 fx_2`)` behaves the same as `(fxarithmetic-shift` fx_1 `(fixnum-` fx_2`))`.

`(fxrotate-bit-field` fx_1 fx_2 fx_3 fx_4`)` procedure

Fx_2, fx_3, and fx_4 must be non-negative and less than `(fixnum-width)`. Fx_4 must be less than the difference between fx_3 and fx_3. The `fxrotate-bit-field` procedure returns the same result as the `fixnum-rotate-bit-field` procedure.

`(fxreverse-bit-field` fx_1 fx_2 fx_3`)` procedure

Fx_2 and fx_3 must be non-negative and less than `(fixnum-width)`. Moreover, fx_2 must be less than or equal to fx_3. The `fxreverse-bit-field` procedure returns the same result as the `fixnum-reverse-bit-field` procedure.

16.4. Flonums

This section describes the `(r6rs arithmetic flonum)` library.

This section uses fl, fl_1 and fl_2 as parameter names for arguments that must be flonums, and ifl, ifl_1 and ifl_2 as parameter names for arguments that must be integer-valued flonums, i.e. flonums for which the `integer-valued?` predicate returns true.

`(flonum?` *obj*`)` procedure

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

`(fl=?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl<?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl<=?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl>?` fl_1 fl_2 fl_3 ...`)` procedure
`(fl>=?` fl_1 fl_2 fl_3 ...`)` procedure

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

```
(fl= +inf.0 +inf.0)          ⟹  #t
(fl= -inf.0 +inf.0)          ⟹  #f
(fl= -inf.0 -inf.0)          ⟹  #t
(fl= 0.0 -0.0)               ⟹  #t
(fl< 0.0 -0.0)               ⟹  #f
(fl= +nan.0 fl)              ⟹  #f
(fl< +nan.0 fl)              ⟹  #f
```

`(flinteger?` fl`)` procedure
`(flzero?` fl`)` procedure
`(flpositive?` fl`)` procedure
`(flnegative?` fl`)` procedure
`(flodd?` ifl`)` procedure
`(fleven?` ifl`)` procedure
`(flfinite?` fl`)` procedure
`(flinfinite?` fl`)` procedure
`(flnan?` fl`)` procedure

These numerical predicates test a flonum for a particular property, returning `#t` or `#f`. The `flinteger?` procedure tests it if the number is an integer, `flzero?` tests if it is `fl=?` to zero, `flpositive?` tests if it is greater than zero, `flnegative?` tests if it is less than zero, `flodd?` tests if it is odd, `fleven?` tests if it is even, `flfinite?` tests if it is not an infinity and not a NaN, `flinfinite?` tests if it is an infinity, and `flnan?` tests if it is a NaN.

```
(flnegative? -0.0)           ⟹ #f
(flfinite? +inf.0)           ⟹ #f
(flfinite? 5.0)              ⟹ #t
(flinfinite? 5.0)            ⟹ #f
(flinfinite? +inf.0)         ⟹ #t
```

Note: `(flnegative? -0.0)` must return `#f`, else it would lose the correspondence with `(fl< -0.0 0.0)`, which is `#f` according to the IEEE standards.

`(flmax` fl_1 fl_2 ...`)` procedure
`(flmin` fl_1 fl_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

`(fl+` fl_1 ...`)` procedure
`(fl*` fl_1 ...`)` procedure

These procedures return the flonum sum or product of their flonum arguments. In general, they should return the flonum that best approximates the mathematical sum or product. (For implementations that represent flonums as IEEE binary floating point numbers, the meaning of "best" is 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
```

`(fl-` fl_1 fl_2 ...`)` procedure
`(fl-` fl`)` procedure
`(fl/` fl_1 fl_2 ...`)` procedure
`(fl/` fl`)` procedure

With two or more arguments, these procedures return the flonum difference or quotient of their flonum arguments, associating to the left. With one argument, however, they return the additive or multiplicative flonum inverse of their argument. In general, they should return the flonum that best approximates the mathematical difference or quotient. (For implementations that represent flonums as IEEE binary floating point numbers, the meaning of "best" is reasonably well-defined by the IEEE standards.)

```
(fl- +inf.0 +inf.0)          ⟹  +nan.0
```

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

```
(fl/ 1.0 0.0)                ⟹ +inf.0
(fl/ -1.0 0.0)               ⟹ -inf.0
(fl/ 0.0 0.0)                ⟹ +nan.0
```

`(flabs` fl`)` procedure

Returns the absolute value of fl.

`(fldiv+mod` fl_1 fl_2`)` procedure
`(fldiv` fl_1 fl_2`)` procedure
`(flmod` fl_1 fl_2`)` procedure
`(fldiv0+mod0` fl_1 fl_2`)` procedure
`(fldiv0` fl_1 fl_2`)` procedure
`(flmod0` fl_1 fl_2`)` procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in section 16.2.1. For zero divisors, these procedures may return a NaN or some meaningless flonum.

```
(fldiv fl1 fl2)              ⟹ fl1 div fl2
(flmod fl1 fl2)              ⟹ fl1 mod fl2
(fldiv+mod fl1 fl2)
          ⟹ fl1 div fl2, fl1 mod fl2
                             ; two return values
(fldiv0 fl1 fl2)             ⟹ fl1 div0 fl2
(flmod0 fl1 fl2)             ⟹ fl1 mod0 fl2
(fldiv0+mod0 fl1 fl2)
          ⟹ fl1 div0 fl2, fl1 mod0 fl2
                             ; two return values
```

`(flnumerator` fl`)` procedure
`(fldenominator` fl`)` procedure

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

```
(flnumerator +inf.0)         ⟹ +inf.0
(flnumerator -inf.0)         ⟹ -inf.0
(fldenominator +inf.0)       ⟹ 1.0
(fldenominator -inf.0)       ⟹ 1.0
(flnumerator 0.75)           ⟹ 3.0 ; example
(fldenominator 0.75)         ⟹ 4.0 ; example
```

The following behavior is strongly recommended but not required:

```
(flnumerator -0.0)           ⟹ -0.0
```

`(flfloor` fl`)` procedure
`(flceiling` fl`)` procedure
`(fltruncate` fl`)` procedure
`(flround` fl`)` procedure

These procedures return integral flonums for flonum arguments that are not infinities or NaNs. For such arguments, `flfloor` returns the largest integral flonum not larger than fl. The `flceiling` procedure returns the smallest integral flonum not smaller than fl. The `fltruncate` procedure returns the integral flonum closest to fl whose absolute value is not larger than the absolute value of fl. The `flround` procedure returns the closest integral flonum to fl, rounding to even when fl is halfway between two integers.

Rationale: The `flround` procedure rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

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

```
(flfloor +inf.0)             ⟹ +inf.0
(flceiling -inf.0)           ⟹ -inf.0
(fltruncate +nan.0)          ⟹ +nan.0
```

`(flexp` fl`)` procedure
`(fllog` fl`)` procedure
`(fllog` fl_1 fl_2`)` procedure
`(flsin` fl`)` procedure
`(flcos` fl`)` procedure
`(fltan` fl`)` procedure
`(flasin` fl`)` procedure
`(flatan` fl`)` procedure
`(flatan` fl_1 fl_2`)` procedure

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

See section 16.2.2 for the underlying mathematical operations. In the event that these operations do not yield a real

result for the given arguments, the result may be a NaN, or may be some meaningless flonum.

Implementations that use IEEE binary floating point arithmetic are encouraged to follow the relevant standards for these procedures.

```
(flexp +inf.0)              ⟹ +inf.0
(flexp -inf.0)              ⟹ 0.0
(fllog +inf.0)              ⟹ +inf.0
(fllog 0.0)                 ⟹ -inf.0
(fllog -0.0)                ⟹ unspecified
          ; if -0.0 is distinguished
(fllog -inf.0)              ⟹ +nan.0
(flatan -inf.0)
          ⟹ -1.5707963267948965
          ; approximately
(flatan +inf.0)
          ⟹ 1.5707963267948965
          ; approximately
```

`(flsqrt` *fl*`)` procedure

Returns the principal square root of *fl*. For a negative argument, the result may be a NaN, or may be some meaningless flonum.

```
(flsqrt +inf.0)             ⟹  +inf.0
```

`(flexpt` fl_1 fl_2`)` procedure

Returns fl_1 raised to the power fl_2. fl_1 should be nonnegative; if fl_1 is negative, then the result may be a NaN, or may be some meaningless flonum. If fl_1 is zero, then the result is zero. For positive fl_1,

$$fl_1^{fl_2} = e^{fl_2 \log fl_1}$$

`&no-infinities` condition type
`(no-infinities?` *obj*`)` procedure
`&no-nans` condition type
`(no-nans?` *obj*`)` procedure

These condition types could be defined by the following code:

```
(define-condition-type &no-infinities
    &implementation-restriction
  no-infinities?)

(define-condition-type &no-nans
    &implementation-restriction
  no-nans?)
```

These types describe that a program has executed an arithmetic operations that is specified to return an infinity or a NaN, respectively, on a Scheme implementation that is not able to represent the infinity or NaN. (See section 16.1.)

`(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.

16.5. Exact arithmetic

This section describes the `(r6rs arithmetic exact)` library.

The exact arithmetic provides generic operations on exact numbers; these operations correspond to their mathematical counterparts. The exact numbers include rationals of arbitrary precision, and exact rectangular complex numbers. A rational number with a denominator of 1 is indistinguishable from its numerator. An exact rectangular complex number with a zero imaginary part is indistinguishable from its real part.

`(exact-number?` *ex*`)` procedure
`(exact-complex?` *ex*`)` procedure
`(exact-rational?` *ex*`)` procedure
`(exact-integer?` *ex*`)` procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is an exact number of the named type, and otherwise return `#f`. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

This section uses *ex*, ex_1, ex_2, and ex_3 as parameter names for arguments that must be exact complex numbers, *ef*, ef_1, ef_2, and ef_3 as parameter names for arguments that must be exact rational numbers, and *ei*, ei_1, ei_2, and ei_3 as parameter names that must be exact integers.

`(exact=?` ex_1 ex_2 ex_3 ...`)` procedure
`(exact>?` ef_1 ef_2 ef_3 ...`)` procedure
`(exact<?` ef_1 ef_2 ef_3 ...`)` procedure
`(exact>=?` ef_1 ef_2 ef_3 ...`)` procedure
`(exact<=?` ef_1 ef_2 ef_3 ...`)` procedure

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

`(exact-zero?` *ex*`)` procedure
`(exact-positive?` *ef*`)` procedure
`(exact-negative?` *ef*`)` procedure
`(exact-odd?` *ei*`)` procedure
`(exact-even?` *ei*`)` procedure

These numerical predicates test an exact number for a particular property, returning `#t` or `#f`. The `exact-zero?` procedure tests if the number is `exact=?` to zero, `exact-positive?` tests if it is greater than zero, `exact-negative?` tests if it is less than zero, `exact-odd?` tests if it is odd, `exact-even?` tests if it is even.

`(exact-max` ef_1 ef_2 ...`)` procedure
`(exact-min` ef_1 ef_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

`(exact+` ex_1 ex_2 ...`)` procedure
`(exact*` ex_1 ex_2 ...`)` procedure

These procedures return the sum or product of their arguments.

`(exact-` ex_1 ex_2 ...`)` procedure
`(exact-` *ex*`)` procedure
`(exact/` ex_1 ex_2 ...`)` procedure
`(exact/` *ex*`)` procedure

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument. The `exact/` procedure raises an exception with condition type `&contract` if a divisor is 0.

`(exact-abs` *ef*`)` procedure

Returns the absolute value of *ef*.

`(exact-div+mod` ef_1 ef_2`)` procedure
`(exact-div` ef_1 ef_2`)` procedure
`(exact-mod` ef_1 ef_2`)` procedure
`(exact-div0+mod0` ef_1 ef_2`)` procedure
`(exact-div0` ef_1 ef_2`)` procedure
`(exact-mod0` ef_1 ef_2`)` procedure

Ef_2 must be nonzero. These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in section 16.2.1.

```
(exact-div ef1 ef2)          ⟹ ef1 div ef2
(exact-mod ef1 ef2)          ⟹ ef1 mod ef2
(exact-div+mod ef1 ef2)
        ⟹ ef1 div ef2, ef1 mod ef2
                          ; two return values
```

```
(exact-div0 ef1 ef2)         ⟹ ef1 div0 ef2
(exact-mod0 ef1 ef2)         ⟹ ef1 mod0 ef2
(exact-div0+mod0 ef1 ef2)
        ⟹ ef1 div0 ef2, ef1 mod0 ef2
                          ; two return values
```

`(exact-gcd` ei_1 ei_2 ...`)` procedure
`(exact-lcm` ei_1 ei_2 ...`)` procedure

These procedures return the greatest common divisor or least common multiple of their arguments.

`(exact-numerator` *ef*`)` procedure
`(exact-denominator` *ef*`)` procedure

These procedures return the numerator or denominator of their argument. The result is computed as if the argument were represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

`(exact-floor` *ef*`)` procedure
`(exact-ceiling` *ef*`)` procedure
`(exact-truncate` *ef*`)` procedure
`(exact-round` *ef*`)` procedure

These procedures return exact integers. The `exact-floor` procedure returns the largest integer not larger than *ef*. The `exact-ceiling` procedure returns the smallest integer not smaller than *ef*. The `exact-truncate` procedure returns the integer closest to *ef* whose absolute value is not larger than the absolute value of *ef*. The `exact-round` procedure returns the closest integer to *ef*, rounding to even when *ef* is halfway between two integers.

`(exact-expt` ef_1 ei_2`)` procedure

Returns ef_1 raised to the power ei_2. 0^{ei} is 1 if $ei = 0$ and 0 if *ei* is positive. If ef_1 is zero and ei_2 is negative, this procedure raises an exception with condition type `&contract`.

`(exact-make-rectangular` ef_1 ef_2`)` procedure
`(exact-real-part` *ex*`)` procedure
`(exact-imag-part` *ex*`)` procedure

The arguments of `exact-make-rectangular` must be exact rationals. Suppose *ex* is a complex number such that

$$ex = ef_1 + ef_2 i.$$

Then:

```
(exact-make-rectangular ef1 ef2)
        ⟹ ex
(exact-real-part ex)         ⟹ ef1
(exact-imag-part ex)         ⟹ ef2
```

`(exact-sqrt` *ei*`)` procedure

Ei must be non-negative. The `exact-sqrt` procedure returns two non-negative exact integers s and r where $ei = s^2 + r$ and $ei < (s+1)^2$.

`(exact-not` *ei*`)` procedure

Returns the exact integer whose two's complement representation is the one's complement of the two's complement representation of *ei*.

`(exact-and` ei_1 ...`)` procedure
`(exact-ior` ei_1 ...`)` procedure
`(exact-xor` ei_1 ...`)` procedure

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

`(exact-if` ei_1 ei_2 ei_3`)` procedure

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

```
(exact-ior (exact-and ei1 ei2)
           (exact-and (exact-not ei1) ei3))
```

`(exact-bit-count` *ei*`)` procedure

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

`(exact-length` *ei*`)` procedure

These procedures return the exact integer that is the result of the following computation:

```
(do ((result 0 (+ result 1))
     (bits (if (exact-negative? ei)
               (exact-not ei)
               ei)
           (exact-arithmetic-shift bits -1)))
    ((exact-zero? bits)
     result))
```

`(exact-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.

```
(exact-first-bit-set 0)     ⟹  -1
(exact-first-bit-set 1)     ⟹  0
(exact-first-bit-set -4)    ⟹  2
```

`(exact-bit-set?` ei_1 ei_2`)` procedure

Ei_2 must be non-negative.

Otherwise returns the result of the following computation:

```
(not (exact-zero?
      (exact-and
       (exact-arithmetic-shift-left 1 ei2)
       ei1)))
```

`(exact-copy-bit` ei_1 ei_2 ei_3`)` procedure

Ei_2 must be non-negative, and ei_3 must be either 0 or 1. The `exact-copy-bit` procedure returns the result of the following computation:

```
(let* ((mask (exact-arithmetic-shift-left 1 ei2)))
  (exact-if mask
            (exact-arithmetic-shift-left ei3 ei2)
            ei1))
```

`(exact-bit-field` ei_1 ei_2 ei_3`)` procedure

Ei_2 and ei_3 must be non-negative, and ei_2 must be less than or equal to ei_3. This procedure returns the result of the following computation:

```
(let* ((mask
        (exact-not
         (exact-arithmetic-shift-left -1 ei3))))
  (exact-arithmetic-shift-right
   (exact-and ei1 mask)
   ei2))
```

`(exact-copy-bit-field` ei_1 ei_2 ei_3 ei_4`)` procedure

Ei_2 and ei_3 must be non-negative, and ei_2 must be less than or equal to ei_3. The `exact-copy-bit-field` procedure returns the result of the following computation:

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

`(exact-arithmetic-shift` ei_1 ei_2`)` procedure

Returns the exact integer result of the following computation:

```
(exact-floor (* ei1 (expt 2 ei2)))
```

`(exact-arithmetic-shift-left` ei_1 ei_2`)` procedure
`(exact-arithmetic-shift-right` ei_1 ei_2`)` procedure

Ei_2 must be non-negative. `exact-arithmetic-shift-left` returns the same result as `exact-arithmetic-shift`, and `(exact-arithmetic-shift-right` ei_1 ei_2`)` returns the same result as `(exact-arithmetic-shift` ei_1 `(exact-` ei_2`))`.

`(exact-rotate-bit-field` ei_1 ei_2 ei_3 ei_4`)` procedure

Ei_2, ei_3, ei_4 must be non-negative, and ei_4 must be less than the difference between ei_2 and ei_3. The procedure returns the result of the following computation:

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

`(exact-reverse-bit-field` ei_1 ei_2 ei_3`)` procedure

Ei_2 and ei_3 must be non-negative, and ei_2 must be less than or equal to ei_3. The `exact-reverse-bit-field` procedure returns the result obtained from the ei_1 by reversing the bit field specified by ei_2 and ei_3.

```
(exact-reverse-bit-field #b1010010 1 4)
          ⟹  88 ; #b1011000
(exact-reverse-bit-field #1010010 91 -4)
          ⟹  &contract exception
```

16.6. Inexact arithmetic

This section describes the `(r6rs arithmetic inexact)` library.

The inexact arithmetic provides generic operations on inexact numbers. The inexact numbers include inexact reals and inexact complex numbers, both of which are distinguishable from the exact numbers. The inexact complex numbers include the flonums, and the procedures described here behave consistently with the corresponding flonum procedures if passed flonum arguments.

`(inexact-number?` *obj*`)` procedure
`(inexact-complex?` *obj*`)` procedure
`(inexact-real?` *obj*`)` procedure
`(inexact-rational?` *obj*`)` procedure
`(inexact-integer?` *obj*`)` procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is an inexact number of the named type, and otherwise they return `#f`. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

This section uses *in*, in_1, in_2, and in_3 as parameter names for arguments that must be inexact numbers, *ir*, ir_1, ir_2, and ir_3 as parameter names for arguments that must be inexact real numbers, *if*, if_1, if_2, and if_3 as parameter names for arguments that must be inexact rationals, and *ii*, ii_1, ii_2, and ii_3 as parameter names for arguments that must be the inexact integers.

`(inexact=?` in_1 in_2 in_3 ...`)` procedure
`(inexact>?` ir_1 ir_2 ir_3 ...`)` procedure
`(inexact<?` ir_1 ir_2 ir_3 ...`)` procedure
`(inexact>=?` ir_1 ir_2 ir_3 ...`)` procedure
`(inexact<=?` ir_1 ir_2 ir_3 ...`)` procedure

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

`(inexact-zero?` *in*`)` procedure
`(inexact-positive?` *ir*`)` procedure
`(inexact-negative?` *ir*`)` procedure
`(inexact-odd?` *ii*`)` procedure
`(inexact-even?` *ii*`)` procedure
`(inexact-finite?` *in*`)` procedure
`(inexact-infinite?` *in*`)` procedure
`(inexact-nan?` *in*`)` procedure

These numerical predicates test an inexact number for a particular property, returning `#t` or `#f`. The `inexact-zero?` procedure tests if the number is `inexact=?` to zero, `inexact-positive?` tests if it is greater than zero, `inexact-negative?` tests if it is less than zero, `inexact-odd?` tests if it is odd, `inexact-even?` tests if it is even, `inexact-finite?` tests if it is not an infinity and not a NaN, `inexact-infinite?` tests if it is an infinity, and `inexact-nan?` tests if it is a NaN.

`(inexact-max` ir_1 ir_2 ...`)` procedure
`(inexact-min` ir_1 ir_2 ...`)` procedure

These procedures return the maximum or minimum of their arguments.

`(inexact+` in_1 in_2 ...`)` procedure
`(inexact*` in_1 in_2 ...`)` procedure

These procedures return the sum or product of their arguments.

`(inexact-` in_1 in_2 ...`)` procedure
`(inexact-` in`)` procedure
`(inexact/` in_1 in_2 ...`)` procedure
`(inexact/` in`)` procedure

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

`(inexact-abs` in`)` procedure

Returns the absolute value of its argument.

`(inexact-div+mod` ir_1 ir_2`)` procedure
`(inexact-div` ir_1 ir_2`)` procedure
`(inexact-mod` ir_1 ir_2`)` procedure
`(inexact-div0+mod0` ir_1 ir_2`)` procedure
`(inexact-div0` ir_1 ir_2`)` procedure
`(inexact-mod0` ir_1 ir_2`)` procedure

Ir_1 must be neither infinite nor a NaN, and ir_2 must be nonzero. These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in section 16.2.1.

```
(inexact-div ir1 ir2)          ⟹ ir1 div ir2
(inexact-mod ir1 ir2)          ⟹ ir1 mod ir2
(inexact-div+mod ir1 ir2)
          ⟹ ir1 div ir2, ir1 mod ir2
                               ; two return values
(inexact-div0 ir1 ir2)         ⟹ ir1 div0 ir2
(inexact-mod0 ir1 ir2)         ⟹ ir1 mod0 ir2
(inexact-div0+mod0 ir1 ir2)
          ⟹ ir1 div0 ir2, ir1 mod0 ir2
                               ; two return values
```

`(inexact-gcd` ii_1 ii_2 ...`)` procedure
`(inexact-lcm` ii_1 ii_2 ...`)` procedure

These procedures return the greatest common divisor or least common multiple of their arguments.

`(inexact-numerator` if`)` procedure
`(inexact-denominator` if`)` procedure

These procedures return the numerator or denominator of if. The result is computed as if if was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0.0 is defined to be 1.0.

`(inexact-floor` ir`)` procedure
`(inexact-ceiling` ir`)` procedure
`(inexact-truncate` ir`)` procedure
`(inexact-round` ir`)` procedure

These procedures return inexact integers for real arguments that are not infinities or NaNs. For such arguments, `inexact-floor` returns the largest integer not larger than ir. The `inexact-ceiling` procedure returns the smallest integer not smaller than ir. The `inexact-truncate` procedure returns the integer closest to in whose absolute value is not larger than the absolute value of in. The `inexact-round` procedure returns the closest integer to in, rounding to even when in is halfway between two integers.

Rationale: The `inexact-round` procedure rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

Although infinities and NaNs are not integers, these procedures return an infinity when given an infinity as an argument, and a NaN when given a NaN.

`(inexact-exp` in`)` procedure
`(inexact-log` in`)` procedure
`(inexact-log` in_1 in_2`)` procedure
`(inexact-sin` in`)` procedure
`(inexact-cos` in`)` procedure
`(inexact-tan` in`)` procedure
`(inexact-asin` in`)` procedure
`(inexact-atan` in`)` procedure
`(inexact-atan` ir_1 ir_2`)` procedure

These procedures compute the usual transcendental functions. The `inexact-exp` procedure computes the base-e exponential of in. The `inexact-log` procedure with a single argument computes the natural logarithm of in (not the base ten logarithm); `(inexact-log` in_1 in_2`)` computes the base-in_2 logarithm of in_1. The `inexact-asin`, `inexact-acos`, and `inexact-atan` procedures compute arcsine, arccosine, and arctangent, respectively. The two-argument variant of `inexact-atan` computes `(inexact-angle (inexact-make-rectangular` ir_1 ir_2`))` (see below).

See section 16.2.2 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 `+nan.0`, or may be some meaningless inexact number.

`(inexact-sqrt` in`)` procedure

Returns the principal square root of in. For a negative argument, the result may be `+nan.0`, or may be some meaningless inexact number. With log defined as in section 16.2.2, the value of `(inexact-sqrt` in`)` could be expressed as

$$e^{\frac{\log in}{2}}.$$

`(inexact-expt` in_1 in_2`)` procedure

Returns in_1 raised to the power in_2. For nonzero in_1,

$$in_1{}^{in_2} = e^{in_2 \log in_1}$$

0.0^{in} is 1 if $in = 0.0$, and 0.0 if `(inexact-real-part` in`)` is positive. Otherwise, for nonzero in_1, this procedure raises an exception with condition type `&implementation-restriction` or returns an unspecified number.

`(inexact-make-rectangular` ir_1 ir_2`)` procedure
`(inexact-make-polar` ir_1 ir_2`)` procedure
`(inexact-real-part` in`)` procedure
`(inexact-imag-part` in`)` procedure
`(inexact-magnitude` in`)` procedure
`(inexact-angle` in`)` procedure

Suppose ir_1, ir_2, ir_3, and ir_4 are inexact real numbers, and ir is a complex number, such that

$$ir = ir_1 + ir_2 i = ir_3 e^{i ir_4}.$$

Then (inexactly):

```
(inexact-make-rectangular ir1 ir2)
          ⟹ ir
(inexact-make-rectangular ir3 ir4)
          ⟹ ir
(inexact-real-part ir)       ⟹ ir1
(inexact-imag-part ir)       ⟹ ir2
(inexact-magnitude ir)       ⟹ |ir3|
(inexact-angle ir)           ⟹ ir_angle
```

where $-\pi \leq ir_{\text{angle}} \leq \pi$ with $ir_{\text{angle}} = ir_4 + 2\pi n$ for some integer n.

```
(inexact-angle -1.0)         ⟹ π
(inexact-angle -1.0+0.0)     ⟹ π
(inexact-angle -1.0-0.0)     ⟹ -π
          ; if -0.0 is distinguished
```

Moreover, suppose ir_1, ir_2 are such that either ir_1 or ir_2 is an infinity, then

```
(inexact-make-rectangular ir1 ir2)
          ⟹ ir
(inexact-magnitude ir)       ⟹ +inf.0
```

17. syntax-case

The `(r6rs syntax-case)` library provides support for writing low-level macros in a high-level style, with automatic syntax checking, input destructuring, output restructuring, maintenance of lexical scoping and referential transparency (hygiene), and support for controlled identifier capture.

Rationale: While many syntax transformers are succinctly expressed using the high-level `syntax-rules` form, others are difficult or impossible to write, including some that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form, ones that maintain state or read from the file system, and ones that construct new identifiers. The `syntax-case` system [14] described here allows the programmer to write transformers that perform these sorts of transformations, and arbitrary additional transformations, without sacrificing the default enforcement of hygiene or the high-level pattern-based syntax matching and template-based output construction provided by `syntax-rules` (section 9.21).

Because `syntax-case` does not require literals, including quoted lists or vectors, to be copied or even traversed, it may be able to preserve sharing and cycles within and among the constants of a program. It also supports source-object correlation, i.e., the maintenance of ties between the original source code and expanded output, allowing implementations to provide source-level support for debuggers and other tools.

17.1. Hygiene

Barendregt's *hygiene condition* [2] 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 [32] 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 17.4) inside the transformer body or one of the helpers it calls. This is essentially the referential transparency property described by Clinger and Rees [10].

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 17.6.

Operationally, the expander can maintain hygiene with the help of *marks* and *substitutions*. Marks are applied selectively by the expander to the output of each transformer it invokes, and substitutions are applied to the portions of each binding form that are supposed to be within the scope of the bound identifiers. Marks are used to distinguish like-named identifiers that are introduced at different times (either present in the source or introduced into the output of a particular transformer call), and substitutions are used to map identifiers to their expand-time values.

Each time the expander encounters a macro use, it applies an *antimark* to the input form, invokes the associated transformer, then applies a fresh mark to the output. Marks and antimarks cancel, so the portions of the input that appear in the output are effectively left unmarked, while the portions of the output that are introduced are marked with the fresh mark.

Each time the expander encounters a binding form it creates a set of substitutions, each mapping one of the (possibly marked) bound identifiers to information about the binding. (For a `lambda` expression, the expander might map each bound identifier to a representation of the formal parameter in the output of the expander. For a `let-syntax` form, the expander might map each bound identifier to the associated transformer.) These substitutions are applied to the portions of the input form in which the binding is supposed to be visible.

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

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

An algebra that defines how marks and substitutions work more precisely is given in section 2.4 of Oscar Waddell's PhD thesis [52].

17.2. Syntax objects

A *syntax object* is a representation of a Scheme form that contains contextual information about the form in addition to its structure. This contextual information is used by the expander to maintain lexical scoping and may also be used by an implementation to maintain source-object correlation.

Syntax objects may be wrapped or unwrapped. A wrapped syntax object (section 17.1), consists of a *wrap* (section 17.1) and some internal representation of a Scheme form. (The internal representation is unspecified, but is typically a datum value or datum value annotated with source information.) A wrapped syntax object representing an identifier is itself referred to as an identifier; thus, the term *identifier* may refer either to the syntactic entity (symbol, variable, or keyword) or to the concrete representation of the syntactic entity as a syntax object. Wrapped syntax objects are distinct from other types of values.

An unwrapped syntax object is one that is unwrapped, fully or partially, i.e., whose outer layers consist of lists and vectors and whose leaves are either wrapped syntax objects or nonsymbol values.

The term syntax object is used in this document to refer to a syntax object that is either wrapped or unwrapped. More formally, a syntax object is:

- a pair or list of syntax objects,
- a vector of syntax objects,
- a nonlist, nonvector, nonsymbol value, or
- a wrapped syntax object.

The distinction between the terms "syntax object" and "wrapped syntax object" is important. For example, when invoked by the expander, a transformer (section 17.3) must accept a wrapped syntax object but may return any syntax object, including an unwrapped syntax object.

17.3. Transformers

In `define-syntax` (section 9.3), `let-syntax`, and `letrec-syntax` forms (section 9.20), a binding for a syntactic keyword must be a ⟨transformer spec⟩. A ⟨transformer spec⟩ must be an expression that evaluates to a transformer.

A transformer is a *transformation procedure* or a *variable transformer*. A transformation procedure is a procedure that must accept one argument, a wrapped syntax object (section 17.2) representing the input, and return a *syntax object* (section 17.2) representing the output. The procedure is called by the expander whenever a reference to a keyword with which it has been associated is found. If the keyword appears in the first position of a list-structured input form, the transformer receives the entire list-structured form, and its output replaces the entire form. If the keyword is found in any other declaration, definition or expression context, the transformer receives a wrapped syntax object representing just the keyword reference, and its output replaces just the reference. Except with variable transformers (see below), an exception with condition type `&syntax` is raised if the keyword appears on the left-hand side of a `set!` expression.

`(make-variable-transformer` *proc*`)` procedure

Proc must be a procedure that accepts one argument, a wrapped syntax object.

The `make-variable-transformer` procedure creates a *variable transformer*. A variable transformer is like an ordinary transformer except that, if a keyword associated with a variable transformer appears on the left-hand side of a `set!` expression, an exception is not raised. Instead, *proc* is called with a wrapped syntax object representing the entire `set!` expression as its argument, and its return value replaces the entire `set!` expression.

17.4. Parsing input and producing output

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

`(syntax-case` ⟨expression⟩ `(`⟨literal⟩ `...)` ⟨clause⟩ `...)` syntax

Syntax: Each ⟨literal⟩ must be an identifier. Each ⟨clause⟩ must take one of the following two forms.

`(`⟨pattern⟩ ⟨output expression⟩`)`
`(`⟨pattern⟩ ⟨fender⟩ ⟨output expression⟩`)`

⟨Fender⟩ and ⟨output expression⟩ must be ⟨expression⟩s.

A ⟨pattern⟩ is an identifier, constant, or one of the following.

`(`⟨pattern⟩ `...)`
`(`⟨pattern⟩ ⟨pattern⟩ `...` . ⟨pattern⟩`)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...` . ⟨pattern⟩`)`
`#(`⟨pattern⟩ `...)`
`#(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...)`

An ⟨ellipsis⟩ is the identifier "`...`" (three periods).

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

Pattern variables match arbitrary input subforms and are used to refer to elements of the input. It is a syntax violation if the same pattern variable appears more than once in a ⟨pattern⟩.

Underscores also match arbitrary input subforms but are not pattern variables and so cannot be used to refer to those elements. Multiple underscores may appear in a ⟨pattern⟩.

A literal identifier matches an input subform if and only if the input subform is an identifier and either both its occurrence in the input expression and its occurrence in the list of literals have the same lexical binding, or the two identifiers have the same name and both have no lexical binding.

A subpattern followed by an ellipsis can match zero or more elements of the input.

More formally, an input form F matches a pattern P if and only if one of the following holds:

- P is an underscore (_).
- P is a pattern variable.
- P is a literal identifier and F is an equivalent identifier in the sense of `free-identifier=?` (section 17.5).
- P is of the form `(`P_1 `...` P_n`)` and F is a list of n elements that match P_1 through P_n.
- P is of the form `(`P_1 `...` P_n `.` P_x`)` and F is a list or improper list of n or more elements whose first n elements match P_1 through P_n and whose nth cdr matches P_x.
- P is of the form `(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a proper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.

- P is of the form `(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n `.` P_x`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a list or improper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, whose next $n-m$ elements match P_{m+1} through P_n, and whose nth and final cdr matches P_x.
- P is of the form `#(`P_1 `...` P_n`)` and F is a vector of n elements that match P_1 through P_n.
- P is of the form `#(`P_1 `...` P_k P_e ⟨ellipsis⟩ P_{m+1} `...` P_n`)`, where ⟨ellipsis⟩ is the identifier `...` and F is a vector of n or more elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.
- P is a pattern datum (any nonlist, nonvector, nonsymbol datum) and F is equal to P in the sense of the `equal?` procedure.

Semantics: `syntax-case` first evaluates ⟨expression⟩. It then attempts to match the ⟨pattern⟩ from the first ⟨clause⟩ against the resulting value, which is unwrapped as necessary to perform the match. If the pattern matches the value and no ⟨fender⟩ is present, ⟨output expression⟩ is evaluated and its value returned as the value of the `syntax-case` expression. If the pattern does not match the value, `syntax-case` tries the second ⟨clause⟩, then the third, and so on. It is a syntax violation if the value does not match any of the patterns.

If the optional ⟨fender⟩ is present, it serves as an additional constraint on acceptance of a clause. If the ⟨pattern⟩ of a given ⟨clause⟩ matches the input value, the corresponding ⟨fender⟩ is evaluated. If ⟨fender⟩ evaluates to a true value, the clause is accepted; otherwise, the clause is rejected as if the pattern had failed to match the value. Fenders are logically a part of the matching process, i.e., they specify additional matching constraints beyond the basic structure of the input.

Pattern variables contained within a clause's ⟨pattern⟩ are bound to the corresponding pieces of the input value within the clause's ⟨fender⟩ (if present) and ⟨output expression⟩. Pattern variables can be referenced only within `syntax` expressions (see below). Pattern variables occupy the same name space as program variables and keywords.

`(syntax` ⟨template⟩`)` syntax

Note: `#'`⟨template⟩ is equivalent to `(syntax` ⟨template⟩`)`.

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

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

`(`⟨subtemplate⟩ `...)`
`(`⟨subtemplate⟩ `...` `.` ⟨template⟩`)`
`#(`⟨subtemplate⟩ `...)`

A ⟨subtemplate⟩ is a ⟨template⟩ followed by zero or more ellipses.

The value of a `syntax` form is a copy of ⟨template⟩ in which the pattern variables appearing within the template are replaced with the input subforms to which they are bound. Pattern data and identifiers that are not pattern variables or ellipses are copied directly into the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the subtemplate than in the associated subpattern, the input form is replicated as necessary. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis, and for at least one such pattern variable, the subtemplate must be followed by exactly as many ellipses as the subpattern in which the pattern variable appears. (Otherwise, the expander would not be able to determine how many times the subform should be repeated in the output.) It is a syntax violation if the consraints of this paragraph are not met.

A template of the form `(`⟨ellipsis⟩ ⟨template⟩`)` is identical to ⟨template⟩, except that ellipses within the template have no special meaning. That is, any ellipses contained within ⟨template⟩ are treated as ordinary identifiers. In particular, the template `(... ...)` produces a single ellipsis. This allows macro uses to expand into forms containing ellipses.

The output produced by `syntax` is wrapped or unwrapped according to the following rules.

- the copy of `(`⟨t_1⟩ `.` ⟨t_2⟩`)` is a pair if ⟨t_1⟩ or ⟨t_2⟩ contain any pattern variables,
- the copy of `(`⟨t⟩ ⟨ellipsis⟩`)` is a list if ⟨t⟩ contains any pattern variables,
- the copy of `#(`⟨t_1⟩ `...` ⟨t_n⟩`)` is a vector if any of ⟨t_1⟩, ..., ⟨t_n⟩ contain any pattern variables, and
- the copy of any portion of ⟨t⟩ not containing any pattern variables is a wrapped syntax object.

The input subforms inserted in place of the pattern variables are wrapped if and only if the corresponding input subforms are wrapped.

The following definitions of `or` illustrate `syntax-case` and `syntax`. The second is equivalent to the first but uses the the `#'` prefix instead of the full `syntax` form.

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_) (syntax #f)]
      [(_ e) (syntax e)]
      [(_ e1 e2 e3 ...)
       (syntax (let ([t e1])
                 (if t t (or e2 e3 ...))))])))

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

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

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

```
(define p (cons 4 5))
(define-syntax p.car
  (lambda (x)
    (syntax-case x ()
      [(_ . rest) #'((car p) . rest)]
      [_ #'(car p)])))
p.car                          ⟹ 4
(set! p.car 15)                ⟹ &syntax exception
```

```
(define p (cons 4 5))
(define-syntax p.car
  (make-variable-transformer
    (lambda (x)
      (syntax-case x (set!)
        [(set! _ e) #'(set-car! p e)]
        [(_ . rest) #'((car p) . rest)]
        [_ #'(car p)]))))
(set! p.car 15)
p.car                          ⟹ 15
p                              ⟹ (15 5)
```

A derived `identifier-syntax` form that simplifies the definition of identifier macros is described in section 17.8.

17.5. Identifier predicates

`(identifier?` *obj*`)` procedure

Returns `#t` if *obj* is an identifier, i.e., a syntax object representing an identifier, and `#f` otherwise.

The `identifier?` procedure is often used within a fender to verify that certain subforms of an input form are identifiers, as in the definition of `rec`, which creates self-contained recursive objects, below.

```
(define-syntax rec
  (lambda (x)
    (syntax-case x ()
      [(_ x e)
       (identifier? #'x)
       #'(letrec ([x e]) x)])))

(map (rec fact
       (lambda (n)
         (if (= n 0)
             1
             (* n (fact (- n 1))))))
     '(1 2 3 4 5))
         ⟹ (1 2 6 24 120)

(rec 5 (lambda (x) x))         ⟹ &syntax exception
```

The procedures `bound-identifier=?` and `free-identifier=?` each take two identifier arguments and return `#t` if their arguments are equivalent and `#f` otherwise. These predicates are used to compare identifiers according to their *intended use* as free references or bound identifiers in a given context.

`(bound-identifier=?` id_1 id_2`)` procedure

Id_1 and id_2 must be identifiers. The procedure `bound-identifier=?` returns true if and only if a binding for one would capture a reference to the other in the output of the transformer, assuming that the reference appears within the scope of the binding. In general,

two identifiers are `bound-identifier=?` only if both are present in the original program or both are introduced by the same transformer application (perhaps implicitly—see `datum->syntax`). Operationally, two identifiers are considered equivalent by `bound-identifier=?` if and only if they have the same name and same marks (section 17.1).

The `bound-identifier=?` procedure can be used for detecting duplicate identifiers in a binding construct or for other preprocessing of a binding construct that requires detecting instances of the bound identifiers.

`(free-identifier=?` *id*$_1$ *id*$_2$`)` procedure

Id$_1$ and *id*$_2$ must be identifiers. The `free-identifier=?` procedure returns `#t` if and only if the two identifiers would resolve to the same binding if both were to appear in the output of a transformer outside of any bindings inserted by the transformer. (If neither of two like-named identifiers resolves to a binding, i.e., both are unbound, they are considered to resolve to the same binding.) Operationally, two identifiers are considered equivalent by `free-identifier=?` if and only the topmost matching substitution for each maps to the same binding (section 17.1) or the identifiers have the same name and no matching substitution.

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

The following definition of unnamed `let` uses `bound-identifier=?` to detect duplicate identifiers.

```
(define-syntax let
  (lambda (x)
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (let notmem?
                       ([x (car ls)] [ls (cdr ls)])
                   (or (null? ls)
                       (and (not (bound-identifier=?
                                  x (car ls)))
                            (notmem? x (cdr ls)))))
                 (unique-ids? (cdr ls))))))
    (syntax-case x ()
      [(_ ((i v) ...) e1 e2 ...)
       (unique-ids? #'(i ...))
       #'((lambda (i ...) e1 e2 ...) v ...)])))
```

The argument `#'(i ...)` to `unique-ids?` is guaranteed to be a list by the rules given in the description of `syntax` above.

With this definition of `let`:

```
(let ([a 3] [a 4]) (+ a a))
        ⟹ &syntax exception
```

However,

```
(let-syntax
  ([dolet (lambda (x)
            (syntax-case x ()
              [(_ b)
               #'(let ([a 3] [b 4]) (+ a b))]))])
  (dolet a))
        ⟹ 7
```

since the identifier `a` introduced by `dolet` and the identifier `a` extracted from the input form are not `bound-identifier=?`.

The following definition of `case` is equivalent to the one in section 17.4. Rather than including `else` in the literals list as before, this version explicitly tests for `else` using `free-identifier=?`.

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

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

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

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

17.6. Syntax-object and datum conversions

`(syntax->datum` *syntax-object*`)` procedure

The procedure `syntax->datum` strips all syntactic information from a syntax object and returns the corresponding Scheme datum.

Identifiers stripped in this manner are converted to their symbolic names, which can then be compared with `eq?`.

Thus, a predicate `symbolic-identifier=?` might be defined as follows.

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

(`datum->syntax` *template-id datum*) procedure

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

The `datum->syntax` procedure allows a transformer to "bend" lexical scoping rules by creating *implicit identifiers* that behave as if they were present in the input form, thus permitting the definition of macros that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form. For example, the following defines a `loop` expression that uses this controlled form of identifier capture to bind the variable `break` to an escape procedure within the loop body. (The derived `with-syntax` form is like `let` but binds pattern variables—see section 17.8.)

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

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

Were `loop` to be defined as

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

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

The datum argument *datum* may also represent an arbitrary Scheme form, as demonstrated by the following definition of `include`.

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

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

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

evaluates to `50`.

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

Using `datum->syntax`, it is even possible to break hygiene entirely and write macros in the style of old Lisp macros. The `lisp-transformer` procedure defined below creates a transformer that converts its input into a datum, calls the programmer's procedure on this datum, and converts the result back into a syntax object that is scoped at top level (or, more accurately, wherever `lisp-transformer` is defined).

```
(define lisp-transformer
  (lambda (p)
    (lambda (x)
      (datum->syntax #'lisp-transformer
        (p (syntax->datum x))))))
```

Using `lisp-transformer`, defining a basic version of Common Lisp's `defmacro` is a straightforward exercise.

17.7. Generating lists of temporaries

Transformers can introduce a fixed number of identifiers into their output simply by naming each identifier. In

some cases, however, the number of identifiers to be introduced depends upon some characteristic of the input expression. A straightforward definition of `letrec`, for example, requires as many temporary identifiers as there are binding pairs in the input expression. The procedure `generate-temporaries` is used to construct lists of temporary identifiers.

`(generate-temporaries` *l*`)` procedure

L must be be a list or syntax object representing a list-structured form; its contents are not important. The number of temporaries generated is the number of elements in *l*. Each temporary is guaranteed to be unique, i.e., different from all other identifiers.

A definition of `letrec` that uses `generate-temporaries` is shown below.

```
(define-syntax letrec
  (lambda (x)
    (syntax-case x ()
      ((_ ((i v) ...) e1 e2 ...)
       (with-syntax
            (((t ...)
              (generate-temporaries (syntax (i ...)))))
         (syntax (let ((i #f) ...)
                   (let ((t v) ...)
                     (set! i t) ...
                     (let () e1 e2 ...)))))))))
```

Any transformer that uses `generate-temporaries` in this fashion can be rewritten to avoid using it, albeit with a loss of clarity. The trick is to use a recursively defined intermediate form that generates one temporary per expansion step and completes the expansion after enough temporaries have been generated. See the definition for `letrec` in appendix B.

17.8. Derived forms and procedures

The forms and procedures described in this section are *derived*, i.e., they can defined in terms of the forms and procedures described in earlier sections of this document.

The definitions of `p.car` in section 17.4 demonstrated how identifier macros might be written using `syntax-case`. Many identifier macros can be defined more succinctly using the derived `identifier-syntax` form.

`(identifier-syntax` ⟨template⟩`)` syntax

`(identifier-syntax (`⟨id$_1$⟩ ⟨template$_1$⟩`)` syntax
`((set!` ⟨id$_2$⟩ ⟨pattern⟩`)`
⟨template$_2$⟩`))`

Syntax: The ⟨id⟩s must be identifiers.

Semantics: When a keyword is bound to a transformer produced by the first form of `identifier-syntax`, references to the keyword within the scope of the binding are replaced by ⟨template⟩.

```
(define p (cons 4 5))
(define-syntax p.car (identifier-syntax (car p)))
p.car                          ⟹ 4
(set! p.car 15)                ⟹ &syntax exception
```

The second, more general, form of `identifier-syntax` permits the transformer to determine what happens when `set!` is used. In this case, uses of the identifier by itself are replaced by ⟨template$_1$⟩, and uses of `set!` with the identifier are replaced by ⟨template$_2$⟩.

```
(define p (cons 4 5))
(define-syntax p.car
  (identifier-syntax
    [_ (car p)]
    [(set! _ e) (set-car! p e)]))
(set! p.car 15)
p.car                          ⟹ 15
p                              ⟹ (15 5)
```

The `identifier-syntax` form may be defined in terms of `syntax-case`, `syntax`, and `make-variable-transformer` as follows.

```
(define-syntax identifier-syntax
  (syntax-rules (set!)
    [(_ e)
     (lambda (x)
       (syntax-case x ()
         [id (identifier? #'id) #'e]
         [(_ x (... ...)) #'(e x (... ...))]))]
    [(_ (id exp1) ((set! var val) exp2))
     (and (identifier? #'id) (identifier? #'var))
     (make-variable-transformer
       (lambda (x)
         (syntax-case x (set!)
           [(set! var val) #'exp2]
           [(id x (... ...)) #'(exp1 x (... ...))]
           [id (identifier? #'id) #'exp1])))]))
```

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

The derived `with-syntax` form is used to bind pattern variables, just as `let` is used to bind variables. This allows a transformer to construct its output in separate pieces, then put the pieces together.

Each ⟨pattern⟩ is identical in form to a `syntax-case` pattern. The value of each ⟨expression⟩ is computed and destructured according to the corresponding ⟨pattern⟩, and pattern variables within the ⟨pattern⟩ are bound as with `syntax-case` to the corresponding portions of the value within ⟨body⟩.

The with-syntax form may be defined in terms of syntax-case as follows.

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((_ ((p e0) ...) e1 e2 ...)
       (syntax (syntax-case (list e0 ...) ()
                 ((p ...) (let () e1 e2 ...))))))))
```

The following definition of cond demonstrates the use of with-syntax to support transformers that employ recursion internally to construct their output. It handles all cond clause variations and takes care to produce one-armed if expressions where appropriate.

```
(define-syntax cond
  (lambda (x)
    (syntax-case x ()
      [(_ c1 c2 ...)
       (let f ([c1 #'c1] [c2* #'(c2 ...)])
         (syntax-case c2* ()
           [()
            (syntax-case c1 (else =>)
              [(else e1 e2 ...) #'(begin e1 e2 ...)]
              [(e0) #'(let ([t e0]) (if t t))]
              [(e0 => e1)
               #'(let ([t e0]) (if t (e1 t)))]
              [(e0 e1 e2 ...)
               #'(if e0 (begin e1 e2 ...))])]
           [(c2 c3 ...)
            (with-syntax ([rest (f #'c2 #'(c3 ...))])
              (syntax-case c1 (=>)
                [(e0) #'(let ([t e0]) (if t t rest))]
                [(e0 => e1)
                 #'(let ([t e0]) (if t (e1 t) rest))]
                [(e0 e1 e2 ...)
                 #'(if e0
                       (begin e1 e2 ...)
                       rest)]))])))]))
```

(quasisyntax ⟨template⟩) syntax

Note: #`⟨template⟩ is equivalent to (quasisyntax ⟨template⟩), #,⟨template⟩ is equivalent to (unsyntax ⟨template⟩), and #,@⟨template⟩ is equivalent to (unsyntax-splicing ⟨template⟩).

The quasisyntax form is similar to syntax, but it allows parts of the quoted text to be evaluated, in a manner similar to the operation of quasiquote (section 9.19).

Within a quasisyntax *template*, subforms of unsyntax and unsyntax-splicing forms are evaluated, and everything else is treated as ordinary template material, as with syntax. The value of each unsyntax subform is inserted into the output in place of the unsyntax form, while the value of each unsyntax-splicing subform is spliced into the surrounding list or vector structure. Uses of unsyntax and unsyntax-splicing are valid only within quasisyntax expressions.

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

The quasisyntax keyword can be used in place of with-syntax in many cases. For example, the definition of case shown under the description of with-syntax above can be rewritten using quasisyntax as follows.

```
(define-syntax case
  (lambda (x)
    (syntax-case x ()
      [(_ e c1 c2 ...)
       #`(let ([t e])
           #,(let f ([c1 #'c1] [cmore #'(c2 ...)])
               (if (null? cmore)
                   (syntax-case c1 (else)
                     [(else e1 e2 ...)
                      #'(begin e1 e2 ...)]
                     [((k ...) e1 e2 ...)
                      #'(if (memv t '(k ...))
                            (begin e1 e2 ...))])
                   (syntax-case c1 ()
                     [((k ...) e1 e2 ...)
                      #`(if (memv t '(k ...))
                            (begin e1 e2 ...)
                            #,(f (car cmore)
                                 (cdr cmore)))]))))])))
```

Uses of unsyntax and unsyntax-splicing with zero or more than one subform are valid only in splicing (list or vector) contexts. (unsyntax *template* ...) is equivalent to (unsyntax *template*) ..., and (unsyntax-splicing *template* ...) is equivalent to (unsyntax-splicing *template*) .... These forms are primarily useful as intermediate forms in the output of the quasisyntax expander.

Note: Uses of unsyntax and unsyntax-splicing with zero or more than one subform enable certain idioms [4], such as #,@#,@, which has the effect of a doubly indirect splicing when used within a doubly nested and doubly evaluated quasisyntax expression, as with the nested quasiquote examples shown in section 9.19.

Note: Any syntax-rules form can be expressed with syntax-case by making the lambda expression and syntax expressions explicit, and syntax-rules may be defined in terms of syntax-case as follows.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      [(_ (k ...) [(_ . p) f ... t] ...)
       #'(lambda (x)
           (syntax-case x (k ...)
             [(_ . p) f ... #'t] ...))])))
```

A more robust implementation would verify that the literals ⟨literal⟩ ... are all identifiers, that the first position of each pattern is an identifier, and that at most one fender is present in each clause.

17.9. Syntax violations

`(syntax-violation` *who message form*`)` procedure
`(syntax-violation` *who message form subform*`)` procedure

Name must be `#f` or a string or a symbol. *Message* must be a string. *Form* must be a syntax object or a datum value. *Subform* must be a syntax object or a datum value. The `syntax-violation` procedure raises an exception, reporting a syntax violation. The *who* argument should describe the macro transformer that detected the exception. The *message* argument should describe the violation. The *form* argument is the erroneous source syntax object or a datum value representing a form. The optional *subform* argument is a syntax object or datum value representing a form that more precisely locates the violation.

If *who* is `#f`, `syntax-violation` attempts to infer an appropriate value for the condition object (see below) as follows: When *form* is either an identifier or a list-structured syntax object containing an identifier as its first element, then the inferred value is the identifier's symbol. Otherwise, no value for *who* is provided as part of the condition object.

The condition object provided with the exception (see chapter 14) has the following condition types:

- If *who* is not `#f` or can be inferred, the condition has condition type `&who`, with *who* as the value of the `who` field. In that case, *who* should identify the procedure or entity that detected the exception. If it is `#f`, the condition does not have condition type `&who`.
- The condition has condition type `&message`, with *message* as the value of the `message` field.
- The condition has condition type `&syntax` with *form* as the value of the `form` field, and *subform* as the value of the `subform` field. If *subform* is not provided, the value of the `subform` field is `#f`.

18. Hash tables

The `(r6rs hash-tables)` library provides hash tables. A *hash table* is a data structure that associates keys with values. Any object can be used as a key, provided a *hash function* and a suitable *equivalence function* is available. A hash function is a procedure that maps keys to integers, and must be compatible with the equivalence function, which is a procedure that accepts two keys and returns true if they are equivalent, otherwise returns `#f`. Standard hash tables for arbitrary objects based on the `eq?` and `eqv?` predicates (see section 9.6) are provided. Also, standard hash functions for several types are provided.

This section uses the *hash-table* parameter name for arguments that must be hash tables, and the *key* parameter name for arguments that must be hash-table keys.

18.1. Constructors

`(make-eq-hash-table)` procedure
`(make-eq-hash-table` k`)` procedure

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

`(make-eqv-hash-table)` procedure
`(make-eqv-hash-table` k`)` procedure

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

`(make-hash-table` *hash-function equiv*`)` procedure
`(make-hash-table` *hash-function equiv* k`)` procedure

Hash-function and *equiv* must be procedures. *Hash-function* will be called by other procedures described in this chapter with a key as argument, and must return a non-negative exact integer. *Equiv* will be called by other procedures described in this chapter with two keys as arguments. The `make-hash-table` procedure returns a newly allocated mutable hash table using *hash-function* as the hash function and *equiv* as the equivalence function used to compare keys. If a third argument is given, the initial capacity of the hash table is set to approximately k elements.

Both the hash function *hash-function* and the equivalence function *equiv* should behave like pure functions on the domain of keys. For example, the `string-hash` and `string=?` procedures are permissible only if all keys are strings and the contents of those strings are never changed so long as any of them continue to serve as a key in the hash table. Furthermore any pair of values for which the equivalence function *equiv* returns true should be hashed to the same exact integers by *hash-function*.

Note: Hash tables are allowed to cache the results of calling the hash function and equivalence function, so programs cannot

rely on the hash function being called for every lookup or update. Furthermore any hash-table operation may call the hash function more than once.

Rationale: Hash-table lookups are often followed by updates, so caching may improve performance. Hash tables are free to change their internal representation at any time, which may result in many calls to the hash function.

18.2. Procedures

`(hash-table?` *hash-table*`)` procedure

Returns `#t` if *hash-table* is a hash table, otherwise returns `#f`.

`(hash-table-size` *hash-table*`)` procedure

Returns the number of keys contained in *hash-table* as an exact integer.

`(hash-table-ref` *hash-table key default*`)` procedure

Returns the value in *hash-table* associated with *key*. If *hash-table* does not contain an association for *key*, then *default* is returned.

`(hash-table-set!` *hash-table key obj*`)` procedure

Changes *hash-table* to associate *key* with *obj*, adding a new association or replacing any existing association for *key*, and returns the unspecified value.

`(hash-table-delete!` *hash-table key*`)` procedure

Removes any association for *key* within *hash-table*, and returns the unspecified value.

`(hash-table-contains?` *hash-table key*`)` procedure

Returns `#t` if *hash-table* contains an association for *key*, otherwise returns `#f`.

`(hash-table-update!` *hash-table key proc default*`)` procedure

Proc must be a procedure that takes a single argument. The `hash-table-update!` procedure applies *proc* to the value in *hash-table* associated with *key*, or to *default* if *hash-table* does not contain an association for *key*. The *hash-table* is then changed to associate *key* with the result of *proc*.

The behavior of `hash-table-update!` is equivalent to the following code, but may be implemented more efficiently in cases where the implementation can avoid multiple lookups of the same key:

```
(hash-table-set!
 hash-table key
 (proc (hash-table-ref
        hash-table key default)))
```

`(hash-table-fold` *proc hash-table init*`)` procedure

Proc must be a procedure that takes three arguments. For every association in *hash-table*, `hash-table-fold` calls *proc* with the association key, the association value, and an accumulated value as arguments. The accumulated value is *init* for the first invocation of *proc*, and for subsequent invocations of *proc*, it is the return value of the previous invocation of *proc*. The order of the calls to *proc* is indeterminate. The return value of `hash-table-fold` is the value of the last invocation of *proc*. If any side effect is performed on the hash table while a `hash-table-fold` operation is in progress, then the behavior of `hash-table-fold` is unspecified.

`(hash-table-copy` *hash-table*`)` procedure
`(hash-table-copy` *hash-table immutable*`)` procedure

Returns a copy of *hash-table*. If the *immutable* argument is provided and is true, the returned hash table is immutable; otherwise it is mutable.

Rationale: Hash table references may be less expensive with immutable hash tables. Also, a library may choose to export a hash table which cannot be changed by clients.

`(hash-table-clear!` *hash-table*`)` procedure
`(hash-table-clear!` *hash-table k*`)` procedure

Removes all associations from *hash-table* and returns the unspecified value.

If a second argument is given, the current capacity of the hash table is reset to approximately *k* elements.

`(hash-table-keys` *hash-table*`)` procedure

Returns a list of all keys in *hash-table*. The order of the list is unspecified. Equivalent to:

```
(hash-table-fold (lambda (k v a) (cons k a))
                 hash-table
                 '())
```

`(hash-table-values` *hash-table*`)` procedure

Returns a list of all values in *hash-table*. The order of the list is unspecified. Equivalent to:

```
(hash-table-fold (lambda (k v a) (cons v a))
                 hash-table
                 '())
```

18.3. Inspection

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

Returns the equivalence function used by *hash-table* to compare keys. For hash tables created with `make-eq-hash-table` and `make-eqv-hash-table`, returns `eq?` and `eqv?` respectively.

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

Returns the hash function used by *hash-table*. For hash tables created by `make-eq-hash-table` or `make-eqv-hash-table`, `#f` is returned.

Rationale: The `make-eq-hash-table` and `make-eqv-hash-table` constructors are designed to hide their hash function. This allows implementations to use the machine address of an object as its hash value, rehashing parts of the table as necessary whenever the garbage collector moves objects to a different address.

`(hash-table-mutable?` *hash-table*`)` procedure

Returns `#t` if *hash-table* is mutable, otherwise returns `#f`.

18.4. Hash functions

The `equal-hash`, `string-hash`, and `string-ci-hash` procedures of this section are acceptable as hash functions only if the keys on which they are called do not suffer side effects while the hash table remains in use.

`(equal-hash` *obj*`)` procedure

Returns an integer hash value for *obj*, based on its structure and current contents. This hash function is suitable for use with `equal?` as an equivalence function.

`(string-hash` *string*`)` procedure

Returns an integer hash value for *string*, based on its current contents. This hash function is suitable for use with `string=?` as an equivalence function.

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

Returns an integer hash value for *string* based on its current contents, ignoring case. This hash function is suitable for use with `string-ci=?` as an equivalence function.

`(symbol-hash` *symbol*`)` procedure

Returns an integer hash value for *symbol*.

19. Enumerations

This chapter describes the `(r6rs enum)` library for dealing with enumerated values and sets of enumerated values. Enumerated values are represented by ordinary symbols, while finite sets of enumerated values form a separate type, known as the *enumeration sets*. The enumeration sets are further partitioned into sets that share the same *universe* and *enumeration type*. These universes and enumeration types are created by the `make-enumeration` procedure. Each call to that procedure creates a new enumeration type.

This library interprets each enumeration set with respect to its specific universe of symbols and enumeration type. This facilitates efficient implementation of enumeration sets and enables the complement operation.

In the definition of the following procedures, let *enum-set* range over the enumeration sets, which are defined as the subsets of the universes that can be defined using `make-enumeration`.

`(make-enumeration` *list*`)` procedure

List must be a list of symbols. The `make-enumeration` procedure creates a new enumeration type whose universe consists of those symbols (in canonical order of their first appearance in the list) and returns that universe as an enumeration set whose universe is itself and whose enumeration type is the newly created enumeration type.

`(enum-set-universe` *enum-set*`)` procedure

Returns the set of all symbols that comprise the universe of its argument.

`(enum-set-indexer` *enum-set*`)` procedure

Returns a unary procedure that, given a symbol that is in the universe of *enum-set*, returns its 0-origin index within the canonical ordering of the symbols in the universe; given a value not in the universe, the unary procedure returns `#f`.

```
(let* ((e (make-enumeration '(red green blue)))
       (i (enum-set-indexer e)))
  (list (i 'red) (i 'green) (i 'blue) (i 'yellow)))
        ⟹ (0 1 2 #f)
```

The `enum-set-indexer` procedure could be defined as follows (using the `memq` procedure from the `(r6rs lists)` library):

```
(define (enum-set-indexer set)
  (let* ((symbols (enum-set->list
                   (enum-set-universe set)))
         (cardinality (length symbols)))
    (lambda (x)
      (let ((probe (memq x symbols)))
```

```
        (if probe
            (- cardinality (length probe))
            #f)))))
```

`(enum-set-constructor` *enum-set*`)` procedure

Returns a unary procedure that, given a list of symbols that belong to the universe of *enum-set*, returns a subset of that universe that contains exactly the symbols in the list. If any value in the list is not a symbol that belongs to the universe, then the unary procedure raises an exception with condition type `&contract`.

`(enum-set->list` *enum-set*`)` procedure

Returns a list of the symbols that belong to its argument, in the canonical order of the universe of *enum-set*.

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

`(enum-set-member?` *symbol enum-set*`)` procedure
`(enum-set-subset?` *enum-set*$_1$ *enum-set*$_2$`)` procedure
`(enum-set=?` *enum-set*$_1$ *enum-set*$_2$`)` procedure

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

The `enum-set-subset?` procedure returns `#t` if the universe of *enum-set*$_1$ is a subset of the universe of *enum-set*$_2$ (considered as sets of symbols) and every element of *enum-set*$_1$ is a member of its second. It returns `#f` otherwise.

The `enum-set=?` procedure returns `#t` if *enum-set*$_1$ is a subset of *enum-set*$_2$ and vice versa, as determined by the `enum-set-subset?` procedure. This implies that the universes of the two sets are equal as sets of symbols, but does not imply that they are equal as enumeration types. Otherwise, `#f` is returned.

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

`(enum-set-union` *enum-set*$_1$ *enum-set*$_2$`)` procedure
`(enum-set-intersection` *enum-set*$_1$ *enum-set*$_2$`)` procedure
`(enum-set-difference` *enum-set*$_1$ *enum-set*$_2$`)` procedure

Enum-set$_1$ and *enum-set*$_2$ must be enumeration sets that have the same enumeration type. If their enumeration types differ, a `&contract` violation is raised.

The `enum-set-union` procedure returns the union of *enum-set*$_1$ and *enum-set*$_2$. The `enum-set-intersection` procedure returns the intersection of *enum-set*$_1$ and *enum-set*$_2$. The `enum-set-difference` procedure returns the difference of *enum-set*$_1$ and *enum-set*$_2$.

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

          ⟹ ((red blue) (red) (green))
```

`(enum-set-complement` *enum-set*`)` procedure

Returns *enum-set*'s complement with respect to its universe.

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

`(enum-set-projection` *enum-set*$_1$ *enum-set*$_2$`)` procedure

Projects *enum-set*$_1$ into the universe of *enum-set*$_2$, dropping any elements of *enum-set*$_1$ that do not belong to the universe of *enum-set*$_2$. (If *enum-set*$_1$ is a subset of the universe of its second, then no elements are dropped, and the injection is returned.)

```
(let ((e1 (make-enumeration
           '(red green blue black)))
      (e2 (make-enumeration
           '(red black white))))
  (enum-set->list
   (enum-set-projection e1 e2))))
                          ⟹ (red black)
```

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

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

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

⟨Type-name⟩ is an identifier that is bound as a syntactic keyword; ⟨symbol⟩ ... are the symbols that comprise the universe of the enumeration (in order).

(⟨type-name⟩ ⟨symbol⟩) checks at macro-expansion time whether ⟨symbol⟩ is in the universe associated with ⟨type-name⟩. If it is, then (⟨type-name⟩ ⟨symbol⟩) is equivalent to ⟨symbol⟩. It is a syntax violation if it is not.

⟨Constructor-syntax⟩ is an identifier that is bound to a macro that, given any finite sequence of the symbols in the universe, possibly with duplicates, expands into an expression that evaluates to the enumeration set of those symbols.

(⟨constructor-syntax⟩ ⟨symbol⟩ ...) checks at macro-expansion time whether every ⟨symbol⟩ ... is in the universe associated with ⟨type-name⟩. It is a syntax violation if one or more is not. Otherwise

(⟨constructor-syntax⟩ ⟨symbol⟩ ...)

is equivalent to

```
((enum-set-constructor (⟨constructor-syntax⟩))
 (list '⟨symbol⟩ ...)).
```

Example:

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

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

20. Miscellaneous libraries

20.1. `when` and `unless`

This section describes the `(r6rs when-unless)` library.

`(when` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...) syntax
`(unless` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...) syntax

Syntax: ⟨Test⟩ must be an expression. *Semantics:* A `when` expression is evaluated by evaluating the ⟨test⟩ expression. If ⟨test⟩ evaluates to a true value, the remaining ⟨expression⟩s are evaluated in order, and the result(s) of the last ⟨expression⟩ is(are) returned as the result(s) of the entire `when` expression. Otherwise, the `when` expression evaluates to the unspecified value. An `unless` expression is evaluated by evaluating the ⟨test⟩ expression. If ⟨test⟩ evaluates to false, the remaining ⟨expression⟩s are evaluated in order, and the result(s) of the last ⟨expression⟩ is(are) returned as the result(s) of the entire `unless` expression. Otherwise, the `unless` expression evaluates to the unspecified value.

```
(when (> 3 2) 'greater)     ⟹ greater
(when (< 3 2) 'greater)     ⟹ the unspecified value
(unless (> 3 2) 'less)      ⟹ the unspecified value
(unless (< 3 2) 'less)      ⟹ less
```

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

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

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

20.2. `case-lambda`

This section describes the `(r6rs case-lambda)` library.

`(case-lambda` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...) syntax

Syntax: Each ⟨clause⟩ should be of the form

(⟨formals⟩ ⟨body⟩)

⟨Formals⟩ must be as in a `lambda` form (section 9.5.2), ⟨body⟩ must be a body according to section 9.4.

Semantics: A `case-lambda` expression evaluates to a procedure. This procedure, when applied, tries to match its arguments to the ⟨clause⟩s in order. The arguments match a clause if one the following conditions is fulfilled:

- ⟨Formals⟩ has the form (⟨variable⟩ ...) and the number of arguments is the same as the number of formal parameters in ⟨formals⟩.
- ⟨Formals⟩ has the form (⟨variable$_1$⟩ ... ⟨variable$_n$⟩ . ⟨variable$_{n+1}$⟩) and the number of arguments is at least n.

- ⟨Formals⟩ has the form ⟨variable⟩.

For the first clause matched by the arguments, the variables of the ⟨formals⟩ are bound to fresh locations containing the argument values in the same arrangement as with `lambda`.

If the arguments match none of the clauses, an exception with condition type `&contract` is raised.

```
(define foo
  (case-lambda
   (() 'zero)
   ((x) (list 'one x))
   ((x y) (list 'two x y))
   ((a b c d . e) (list 'four a b c d e))
   (rest (list 'rest rest))))

(foo)                    ⟹ zero
(foo 1)                  ⟹ (one 1)
(foo 1 2)                ⟹ (two 1 2)
(foo 1 2 3)              ⟹ (rest (1 2 3))
(foo 1 2 3 4)            ⟹ (four 1 2 3 4 ())
```

A sample definition of `case-lambda` in terms of simpler forms is in appendix B.

20.3. Delayed evaluation

This section describes the `(r6rs promises)` library.

`(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.

See the description of `force` (section 20.3) for a more complete description of `delay`.

`(force` *promise*`)` procedure

Promise must be a promise.

Forces the value of *promise* (see `delay`, section 20.3). If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned.

```
(force (delay (+ 1 2)))     ⟹ 3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                            ⟹ (3 3)

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail a-stream)))
                            ⟹ 2
```

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

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                           ⟹ a promise
(force p)                   ⟹ 6
p                           ⟹ a promise, still
(begin (set! x 10)
       (force p))           ⟹ 6
```

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

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

The expression

`(delay` ⟨expression⟩`)`

has the same meaning as the procedure call

`(make-promise (lambda ()` ⟨expression⟩`))`

as follows

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

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
```

```
(let ((x (proc)))
  (if result-ready?
      result
      (begin (set! result-ready? #t)
             (set! result x)
             result))))))))
```

Rationale: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of `make-promise`.

Various extensions to this semantics of `delay` and `force` are supported in some implementations:

- Calling `force` on an object that is not a promise may simply return the object.
- It may be the case that there is no means by which a promise can be operationally distinguished from its forced value. That is, expressions like the following may evaluate to either `#t` or to `#f`, depending on the implementation:

```
(eqv? (delay 1) 1)           ⟹  unspecified
(pair? (delay (cons 1 2)))   ⟹  unspecified
```

- Some implementations may implement "implicit forcing," where the value of a promise is forced by primitive procedures like `cdr and +`:

```
(+ (delay (* 3 7)) 13)       ⟹  34
```

20.4. Command-line access

The procedure described in this section is exported by the `(r6rs scripts)` library.

`(command-line-arguments)` procedure

When a script is being executed, this returns a list of strings with at least one element. The first element is an implementation-specific name for the running script. The following elements are command-line arguments according to the operating platform's conventions.

21. Composite library

The `(r6rs)` library is a composite of most of the libraries described in this report. The only exceptions are:

- `(r6rs records explicit)` (section 13.2)
- `(r6rs mutable-pairs)` (chapter 23)
- `(r6rs eval)` (chapter 22)
- `(r6rs r5rs)` (chapter 24)

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

Note: Even though `(r6rs records explicit)` is not included, `(r6rs records implicit)` is included, which subsumes the functionality of `(r6rs records explicit)`.

All of the bindings exported by å6rs are exported for both `run` and `expand`; see section 6.2.

22. `eval`

The `(r6rs eval)` library allows a program to create Scheme expressions as data at run time and evaluate them.

`(eval` *expression environment-specifier*`)` procedure

Evaluates *expression* in the specified environment and returns its value. *Expression* must be a valid Scheme expression represented as a datum value, and *environment-specifier* must be a *library specifier*, which can be created using the `environment` procedure described below.

If the first argument to `eval` is not a syntactically correct expression, then `eval` must raise an exception with condition type `&syntax`. Specifically, if the first argument to `eval` is a definition, it must raise an exception with condition type `&eval-definition`.

`(environment` *import-spec ...*`)` procedure

Import-spec must be a datum representing an ⟨import spec⟩ (see section 6.1). The `environment` procedure returns an environment corresponding to *import-spec*

The bindings of the environment represented by the specifier are immutable: If `eval` is applied to an expression that attempts to assign to one of the variables of the environment, `eval` must raise an exception with a condition type `&contract`.

```
(library (foo)
  (export)
  (import (r6rs))
  (write (eval '(let ((x 3)) x) (environment '(r6rs)))
         writes 3

(library foo
  (export)
  (import (r6rs)
  (write
    (eval
      '(eval:car (eval:cons 2 4))
      '(add-prefix (only (r6rs) car cdr cons null?)
```

```
        eval:))))
```
writes `2`

23. Mutable pairs

The procedures provided by the `(r6rs mutable-pairs)` library allow new values to be assigned to the car and cdr fields of previously allocated pairs. In programs that use this library, the criteria for determining the validity of list arguments are more complex than in programs whose lists are immutable. Section 23.2 spells out the definitions that are needed to clarify the specifications of procedures that accept lists. Section 23.3 uses those definitions to clarify the specifications of procedures that are described by this report.

23.1. Procedures

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

Stores *obj* in the car field of *pair*. Returns the unspecified value.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)            ⟹  the unspecified value
(set-car! (g) 3)            ⟹  unspecified
          ; should raise &contract exception
```

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

`(set-cdr!` *pair obj*`)` procedure

Stores *obj* in the cdr field of *pair*. Returns the unspecified value.

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

```
(let ((x (list 'a 'b 'c 'a))
      (y (list 'a 'b 'c 'a 'b 'c 'a)))
  (set-cdr! (list-tail x 2) x)
  (set-cdr! (list-tail y 5) y)
  (list
   (equal? x x)
   (equal? x y)
   (equal? (list x y 'a) (list y x 'b))))
        ⟹  (#t #t #f)
```

23.2. Mutable list arguments

Through the `set-car!` and `set-cdr!` procedures, lists are mutable in Scheme, so a pair that is the head of a list at one moment may not always be the head of a list:

```
(define x (list 'a 'b 'c))
(define y x)
y                           ⟹  (a b c)
(list? y)                   ⟹  #t
(set-cdr! x 4)              ⟹  the unspecified value
x                           ⟹  (a . 4)
(eqv? x y)                  ⟹  #t
y                           ⟹  (a . 4)
(list? y)                   ⟹  #f
(set-cdr! x x)              ⟹  the unspecified value
(list? x)                   ⟹  #f
```

Any procedures defined in this report that are specified to accept a list argument must check if that argument indeed appears to be a list. This checking is complicated by the fact that some such procedures, e.g. `map` and `filter`, call arbitrary procedures that are passed as arguments. These procedures may mutate the list while it is being traversed. Moreover, in the presence of concurrent evaluation, whether a pair is the head of a list is not computable in general.

Consequently, procedures like `length` are only required to confirm that a list argument is a *plausible list*. Informally, a plausible list is an object that appears to be a list during a sequential traversal, where that traversal must also attempt to detect a cycle. In particular, an immutable plausible list is always a list. A more formal definition follows.

Plausible lists are defined with respect to the time interval between the time an argument is passed to the specified procedure and the first return of a value to that procedure's continuation.

Note: In most implementations, the definitions that follow are believed to be invariant under reasonable transformations of global time [6].

A *plausible list up to n between times t_0 and t_n* is a Scheme value x such that

1. x is a pair, and n is 0; or
2. x is the empty list, and n is 0; or
3. x is a pair p, $n > 0$, and there exists some time t_1 in $(t_0, t_n]$ such that taking the cdr of p at time t_1 yields a plausible list up to $n - 1$ between times t_1 and t_n.

A *plausible list up to and including n* is a Scheme value x such that

1. x is a pair, and n is 0; or

2. x is a pair p, $n > 0$, and there exists some time t_1 in $(t_0, t_n]$ such that taking the cdr of p at time t_1 yields a plausible list up to and including $n-1$ between times t_1 and t_n.

A *plausible list of length n between times t_0 and t_n* is a Scheme value x such that

1. x is the empty list, and n is 0; or
2. x is a pair p, $n > 0$, and there exists some time t_1 in $(t_0, t_n]$ such that taking the cdr of p at time t_1 yields a plausible list of length $n-1$ between times t_1 and t_n.

A *plausible prefix of length n between times t_0 and t_n* is a sequence of Scheme values $x_0, \ldots, x_n$ and strictly increasing times $t_1, \ldots, t_n$ such that x_0 through x_{n-1} are pairs, x_n is either the empty list or a pair, and taking the cdr of x_{i-1} at time t_i yields x_i.

A plausible alist up to n between times t_0 and t_n is a Scheme value x such that

1. x is a pair, and n is 0; or
2. x is the empty list, and n is 0; or
3. x is a pair p, $n > 0$, and there exist times t_1 and t_1' in $(t_0, t_n]$ such that the car of x at time t_1' is a pair and the cdr of x at time t_1 is a plausible alist up to $n-1$ between times t_1 and t_n.

A plausible alist of length n is defined similarly, as is *a plausible alist prefix of length n*.

A plausible list (alist) between times t_0 and t_n is a plausible list (alist) of some length n between those times.

23.3. Procedures with list arguments

This section clarifies the domains of procedures in the base library and the `(r6rs lists)` library.

23.3.1. Base-library procedures

These are clarifications to the domains of the procedures of the base library described in sections 9.12, 9.15, and 9.18:

`(list?` *obj*`)` procedure

Returns `#t` if *obj* is a list that is not modified by `set-cdr!` between entry and exit. Returns `#f` if *obj* is not a plausible list. Otherwise, this procedure returns an unspecified boolean or does not return at all.

Note: The unspecified and non-terminating cases can occur only in implementations that allow the argument to be modified by concurrent execution, which is beyond the scope of this document. To reduce the number of unspecified cases that must be mentioned, the rest of this chapter will mostly ignore the possibility of unspecified behavior being caused by concurrent execution.

`(length` *list*`)` procedure

List must be a plausible list.

Note: In other words, an exception must be raised if *list* is not a plausible list. That *list* is a plausible list is not by itself sufficient to avoid the exception, however. If *list* is a plausible list, but is mutated in certain ways during the call to `length`, then an exception may still be raised. This can occur only in implementations that allow concurrent execution. The rest of this chapter will mostly ignore the possibility of exceptions being caused by concurrent modification of an argument.

`(append` *list* ... *obj*`)` procedure

All *list*s must be plausible lists.

`(reverse` *list*`)` procedure

List must be a plausible list.

`(list-tail` *l k*`)` procedure

L must be a plausible list up to *k*.

`(list-ref` *l k*`)` procedure

L must be a plausible list up to and including *k*.

`(map` *proc* $list_1$ $list_2$ ...`)` procedure

Proc must be a procedure; if any of the $list_j$ are nonempty, then *proc* must take as many arguments as there are $list_j$. A natural number n must exist such that all $list_j$ are plausible lists of length n.

Note: In other words, an exception must be raised if no such n exists. The existence of a natural number n such that all of the *list*s are plausible lists of length n is not by itself sufficient to avoid the exception, however. If *proc* mutates the lists in certain ways during the call to `map`, then an exception may still be raised, even in systems that do not allow concurrent execution.

```
(let* ((ones (list 1))
       (f (lambda (x)
            (set-cdr! ones (list x))
            (set! ones (cdr ones))
            2)))
  ; ones is a plausible list
  (map f ones))              ⟹ unspecified
          ; may not terminate
```

`(for-each` *proc* $list_1$ $list_2$ ...`)` procedure

A natural number n must exist such that all $list_j$ are plausible lists of length n.

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

List must be a plausible list where, for every natural number n and for every plausible prefix x_i of that argument of length n, there exists a time t with $t_i < t < t_r$, where t_r is the time of first return from `list->string`, for which the car of x_i is a character.

`(apply` *proc* arg_1 ... *args*`)` procedure

Proc must be a procedure and *args* must be a plausible list.

23.3.2. List utilities

These are clarifications to the domains of the procedures of the `(r6rs lists)` library described in chapter 12:

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

Proc must be a procedure; it must take a single argument if *list* is non-empty. Either there exists a natural number n such that *list* is a plausible list of length n between entry and some subsequent time before exit, or there exists a natural number n, Scheme objects x_j, and times t_j such that $list, x_1, \ldots, x_n$ and $t_1, \ldots, t_n$ is a plausible prefix up to and including n, where t_1 is after entry and t_n is before exit and there exists t' before exit such that $t' > t_n$, the car of x_n at t' is y, and a call to *proc* with argument y at some time after t' but before exit yields a true value.

`(forall` *proc* l_1 l_2 ...`)` procedure
`(exists` *proc* l_1 l_2 ...`)` procedure

Proc must be a procedure; if any l_j is nonempty, then *proc* must take as many arguments as there are ls. Either there exists a natural number n such that every l_i is a plausible list of length n between entry and some subsequent time before exit, or there exists a natural number n, Scheme objects $x_{i,j}$, and times $t_{i,j}$ such that for every l_i: $l_i, x_{i,1}, \ldots, x_{i,n}$ and $t_{i,1}, \ldots, t_{i,n}$ is a plausible prefix up to and including n, where $t_{i,1}$ is after entry and $t_{i,n}$ is before exit and there exist t'_i before exit such that $t'_i > t_{i,n}$ (where $t_{i,0}$ is defined to be t_0), the car of $x_{i,n}$ at t'_i is y_i, and a call to *proc* on the y_i at some time after the maximum of the t'_i but before exit yields a false value (for `forall`) or a true value (for `exists`).

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

Proc must be a procedure; it must take a single argument if *list* is non-empty. *List* must be a plausible list.

`(fold-left` *kons nil* $list_1$ $list_2$ ... $list_n$`)` procedure
`(fold-right` *kons nil* $list_1$ $list_2$ ... $list_n$`)` procedure

Kons must be a procedure; if the *list*s are non-empty, it must take one more argument than there are *list*s. There must exist a natural number n such that every *list* is a plausible list of length n between entry and some subsequent time before exit.

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

Proc must be a procedure; it must take a single argument if *list* is non-empty. *List* must be a plausible list.

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

Proc must be a procedure; if l is nonempty, then it must take a single argument. L must be a plausible list or a value according to the conditions specified below.

If l is not a plausible list, then it must be such that a natural number n exists where l is the first Scheme value of a plausible prefix of length n such that the car of the last value x_n of that prefix satisfies the given condition at some time after t_n and before the procedure returns.

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

Al (for "association list") must be a a plausible alist or a value according to the conditions specified below. *Proc* must be a procedure; if *al* is nonempty, then *proc* must take a single argument.

If *al* is not a plausible alist, then a natural number n must exist such that *al* is the first Scheme value of a plausible prefix of length n such that every Scheme value x_1 through x_n of that prefix is a pair, and x_n has a pair as its car at some time after t_n, and at some time after that the car of that pair is the first argument (or, for `assp`, a value for which *proc* returns true), all before the procedure returns.

24. R^{5}RS compatibility

The procedures described in this chapter are exported from the `(r6rs r5rs)` library, and provide procedures described in the previous revision of this report [28], but omitted from this revision.

`(exact->inexact` z`)` procedure
`(inexact->exact` z`)` procedure

These are the same as the `->inexact` and `->exact` procedures; see section 9.10.2.

`(quotient` n_1 n_2`)` procedure
`(remainder` n_1 n_2`)` procedure
`(modulo` n_1 n_2`)` procedure

These procedures implement number-theoretic (integer) division. n_2 must be non-zero. All three procedures return integers. If n_1/n_2 is an integer:

```
(quotient n1 n2)      ⟹ n1/n2
(remainder n1 n2)     ⟹ 0
(modulo n1 n2)        ⟹ 0
```

If n_1/n_2 is not an integer:

```
(quotient n1 n2)      ⟹ nq
(remainder n1 n2)     ⟹ nr
(modulo n1 n2)        ⟹ nm
```

where n_q is n_1/n_2 rounded towards zero, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r and n_m differ from n_1 by a multiple of n_2, n_r has the same sign as n_1, and n_m has the same sign as n_2.

Consequently, for integers n_1 and n_2 with n_2 not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                              ⟹ #t
```

provided all numbers involved in that computation are exact.

```
(modulo 13 4)                 ⟹ 1
(remainder 13 4)              ⟹ 1

(modulo -13 4)                ⟹ 3
(remainder -13 4)             ⟹ -1

(modulo 13 -4)                ⟹ -3
(remainder 13 -4)             ⟹ 1

(modulo -13 -4)               ⟹ -1
(remainder -13 -4)            ⟹ -1

(remainder -13 -4.0)          ⟹ -1.0  ; inexact
```

Note: These procedures could be defined in terms of `div` and `mod` (see section 9.10.2) as follows (without checking of the argument types):

```
(define (sign n)
  (cond
    ((negative? n) -1)
    ((positive? n) 0)
    (else 0)))

(define (quotient  n1 n2)
  (* (sign n1) (sign n2) (div (abs n1) (abs n2))))

(define (remainder n1 n2)
  (* (sign n1) (mod (abs n1) (abs n2))))

(define (modulo n1 n2)
  (* (sign n2) (mod (* (sign n2) n1) (abs n2))))
```

`(null-environment` n`)` procedure

N must be the exact integer 5. The `null-environment` procedure returns an environment specifier suitable for use with `eval` (see chapter 22) representing an environment that is empty except for the (syntactic) bindings for all syntactic keywords described in the previous revision of this report [28].

`(scheme-report-environment` n`)` procedure

N must be the exact integer 5. The `scheme-report-environment` procedure returns an environment specifier for an environment that is empty except for the bindings for the standard procedures described in the previous revision of this report [28], omitting `load`, `transcript-on`, `transcript-off`, and `char-ready?`. The bindings have as values the procedures of the same names described in this report.

APPENDICES

Appendix A. Formal semantics

This chapter is a placeholder for a formal semantics written using PLT Redex [34, 35]. It will appear in the final version of this report.

Appendix B. Sample definitions for derived forms

This appendix contains sample definitions for some of the keywords described in this report in terms of simpler forms:

`cond`

The `cond` keyword (section 9.5.5) could be defined in terms of `if`, `let` and `begin` using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))
```

`case`

The `case` keyword (section 9.5.5) could be defined in terms of `let`, `cond`, and `memv` (see chapter 12) using `syntax-rules` (see section 9.21) as follows:

```
(define-syntax case
  (syntax-rules (else)
    ((case expr0
       ((key ...) res1 res2 ...)
       ...
       (else else-res1 else-res2 ...))
     (let ((tmp expr0))
       (cond
         ((memv tmp '(key ...)) res1 res2 ...)
         ...
         (else else-res1 else-res2 ...))))
    ((case expr0
       ((keya ...) res1a res2a ...)
       ((keyb ...) res1b res2b ...)
       ...)
     (let ((tmp expr0))
       (cond
         ((memv tmp '(keya ...)) res1a res2a ...)
         ((memv tmp '(keyb ...)) res1b res2b ...)
         ...)))))
```

`letrec`

The `letrec` keyword (section 9.5.6) could be defined approximately in terms of `let` and `set!` using `syntax-rules` (see section 9.21), using a helper to generate the temporary variables needed to hold the values before the assignments are made, as follows:

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec () body1 body2 ...)
     (let () body1 body2 ...))
    ((letrec ((var init) ...) body1 body2 ...)
     (letrec-helper
       (var ...)
       ()
       ((var init) ...)
       body1 body2 ...))))

(define-syntax letrec-helper
  (syntax-rules ()
    ((letrec-helper
       ()
       (temp ...)
       ((var init) ...)
       body1 body2 ...)
     (let ((var <undefined>) ...)
       (let ((temp init) ...)
         (set! var temp)
         ...)
       (let () body1 body2 ...)))
    ((letrec-helper
       (x y ...)
       (temp ...)
       ((var init) ...)
       body1 body2 ...)
     (letrec-helper
       (y ...)
       (newtemp temp ...)
       ((var init) ...)
       body1 body2 ...))))
```

The syntax `<undefined>` represents an expression that returns something that, when stored in a location, causes an exception with condition type `&contract` to be raised if an attempt to read to or write from the location occurs before the assignments generated by the `letrec` transformation take place. (No such expression is defined in Scheme.)

`let-values`

The following definition of `let-values` (section 9.5.6) using `syntax-rules` (see section 9.21) employs a pair of helpers to create temporary names for the formals.

```
(define-syntax let-values
  (syntax-rules ()
    ((let-values (binding ...) body1 body2 ...)
     (let-values-helper1
       ()
       (binding ...)
       body1 body2 ...))))

(define-syntax let-values-helper1
  ;; map over the bindings
  (syntax-rules ()
    ((let-values
       ((id temp) ...)
       ()
       body1 body2 ...)
     (let ((id temp) ...) body1 body2 ...))
    ((let-values
       assocs
       ((formals1 expr1) (formals2 expr2) ...)
       body1 body2 ...)
     (let-values-helper2
       formals1
       ()
       expr1
       assocs
       ((formals2 expr2) ...)
       body1 body2 ...))))

(define-syntax let-values-helper2
  ;; create temporaries for the formals
  (syntax-rules ()
    ((let-values-helper2
       ()
       temp-formals
       expr1
       assocs
       bindings
       body1 body2 ...)
     (call-with-values
       (lambda () expr1)
       (lambda temp-formals
         (let-values-helper1
           assocs
           bindings
           body1 body2 ...))))
    ((let-values-helper2
       (first . rest)
       (temp ...)
       expr1
       (assoc ...)
       bindings
       body1 body2 ...)
     (let-values-helper2
       rest
       (temp ... newtemp)
       expr1
       (assoc ... (first newtemp))
       bindings
       body1 body2 ...))
    ((let-values-helper2
       rest-formal
       (temp ...)
       expr1
       (assoc ...)
       bindings
       body1 body2 ...)
     (call-with-values
       (lambda () expr1)
       (lambda (temp ... . newtemp)
         (let-values-helper1
           (assoc ... (rest-formal newtemp))
           bindings
           body1 body2 ...))))))
```

`case-lambda`

The `case-lambda` keyword (see section 20.2) could be defined in terms of base library by the following macros:

```
(define-syntax case-lambda
  (syntax-rules ()
    ((case-lambda
      (formals-0 body0-0 body1-0 ...)
      (formals-1 body0-1 body1-1 ...)
      ...)
     (lambda args
       (let ((l (length args)))
         (case-lambda-helper
          l args
          (formals-0 body0-0 body1-0 ...)
          (formals-1 body0-1 body1-1 ...) ...))))))

(define-syntax case-lambda-helper
  (syntax-rules ()
    ((case-lambda-helper
      l args
      ((formal ...) body ...)
      clause ...)
     (if (= l (length '(formal ...)))
         (apply (lambda (formal ...) body ...)
                args)
         (case-lambda-helper l args clause ...)))
    ((case-lambda-helper
      l args
      ((formal . formals-rest) body ...)
```

```
      clause ...)
     (case-lambda-helper-dotted
      l args
      (body ...)
      (formal . formals-rest)
      formals-rest 1
      clause ...))
    ((case-lambda-helper
      l args
      (formal body ...))
     (let ((formal args))
      body ...))))

(define-syntax case-lambda-helper-dotted
  (syntax-rules ()
    ((case-lambda-helper-dotted
      l args
      (body ...)
      formals
      (formal . formals-rest) k
      clause ...)
     (case-lambda-helper-dotted
      l args
      (body ...)
      formals
      formals-rest (+ 1 k)
      clause ...))
    ((case-lambda-helper-dotted
      l args
      (body ...)
      formals
      rest-formal k
      clause ...)
     (if (>= l k)
         (apply (lambda formals body ...) args)
         (case-lambda-helper
          l args clause ...)))))
```

Appendix C. Additional material

The Schemers web site at

`http://www.schemers.org/`

as well as the Readscheme site at

`http://library.readscheme.org/`

contain extensive Scheme bibliographies, as well as papers, programs, implementations, and other material related to Scheme.

Appendix D. Example

This section describes an example consisting of the `(runge-kutta)` library, which provides an `integrate-system` procedure that integrates the system

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

As the `(runge-kutta)` library makes use of the `(r6rs base)` and the `(r6rs promises)` libraries, the library skeleton looks as follows:

```
#!r6rs
(library (runge-kutta)
  (export integrate-system)
  (import (r6rs base)
          (r6rs promises))
  ⟨library body⟩)
```

The procedure definitions go in the place of ⟨library body⟩ described below:

The parameter `system-derivative` is a function that takes a system state (a vector of values for the state variables $y_1, \ldots, y_n$) and produces a system derivative (the values $y'_1, \ldots, y'_n$). The parameter `initial-state` provides an initial system state, and `h` is an initial guess for the length of the integration step.

The value returned by `integrate-system` is an infinite stream of system states.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

The `runge-Kutta-4` procedure takes a function, `f`, that produces a system derivative from a system state. The `runge-Kutta-4` procedure produces a function that takes a system state and produces a new system state.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3)))))))))

(define elementwise
  (lambda (f)
    (lambda vectors
```

```
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

The `map-streams` procedure is analogous to `map`: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

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

The following script illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
#! /usr/bin/env scheme-script
#!r6rs
(import (r6rs base)
        (r6rs i/o simple)
        (runge-kutta))

(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01)))

(letrec ((loop (lambda (s)
                 (newline)
                 (write (head s))
                 (loop (tail s)))))
  (loop the-states))

0
```

This prints output like the following:

```
#(1 0)
#(0.99895054 9.994835e-6)
#(0.99780226 1.9978681e-5)
#(0.9965554 2.9950552e-5)
#(0.9952102 3.990946e-5)
#(0.99376684 4.985443e-5)
#(0.99222565 5.9784474e-5)
#(0.9905868 6.969862e-5)
#(0.9888506 7.9595884e-5)
#(0.9870173 8.94753e-5)
```

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] H. P. Barendregt. Introduction to the Lambda Calculus. *Nieuw Archief voor Wisenkunde* 4 (2):337–372, 1984.

[3] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[4] Alan Bawden. Quasiquotation in Lisp. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Technical report BRICS-NS99 -1, University of Aarhus, pages 4–12, 1999.

[5] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116.

[6] William Clinger. Foundations of Actor Semantics. MIT Artificial Intelligence Laboratory Technical Report 633, May 1981.

[7] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[8] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[9] William Clinger and Jonathan Rees, editors. The revised^4 report on the algorithmic language Scheme. In *ACM Lisp Pointers* 4(3), pages 1–55, 1991.

[10] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[11] William Clinger. Proper Tail Recursion and Space Efficiency. To appear in *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, June 1998.

[12] Will Clinger, R. Kent Dybvig, Michael Sperber and Anton van Straaten. SRFI 76: R6RS Records. `http://srfi.schemers.org/srfi-76/`, 2005.

[13] William D Clinger and Michael Sperber. SRFI 77: Preliminary Proposal for R6RS Arithmetic. `http://srfi.schemers.org/srfi-77/`, 2005.

[14] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4):295–326, 1993.

[15] R. Kent Dybvig *The Scheme Programming Language.* Third edition. MIT Press, Cambridge, 2003. `http://www.scheme.com/tspl3/`

[16] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide.* Cadence Research Systems, 2005. `http://www.scheme.com/csug7/`

[17] R. Kent Dybvig SRFI 93: R6RS `syntax-case` macros. `http://srfi.schemers.org/srfi-93/`, 2006.

[18] Sebastian Egner, Richard Kelsey, and Michael Sperber. Cleaning up the tower: Numbers in Scheme. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 109–120, September 22, 2004, Snowbird, Utah. Technical report TR600, `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR600` Computer Science Department, Indiana University.

[19] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [23].

[20] Matthew Flatt. *PLT MzScheme: Language Manual, No. 301.* 2006. `http://download.plt-scheme.org/doc/301/html/mzscheme/`

[21] Matthew Flatt and Kent Dybvig SRFI 83: R6RS Library Syntax. `http://srfi.schemers.org/srfi-83/`, 2005.

[22] Matthew Flatt and Mark Feeley. SRFI 75: R6RS Unicode data. `http://srfi.schemers.org/srfi-75/`, 2005.

[23] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[24] Martin Gasbichler and Michael Sperber SRFI 22: Running Scheme Scripts on Unix. `http://srfi.schemers.org/srfi-22/`, 2002.

[25] Lars T Hansen. SRFI 11: Syntax for receiving multiple values. `http://srfi.schemers.org/srfi-11/`, 2000.

[26] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[27] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[28] Richard Kelsey, William Clinger and Jonathan Rees, editors. The revised^5 report on the algorithmic language Scheme. In *Higher-Order and Symbolic Computation* 11(1), pages 7–105, 1998.

[29] Richard Kelsey and Michael Sperber SRFI 34: Exception Handling for Programs. `http://srfi.schemers.org/srfi-34/`, 2002.

[30] Richard Kelsey and Michael Sperber SRFI 35: Conditions. `http://srfi.schemers.org/srfi-35/`, 2002.

[31] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[32] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161.

[33] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, February 1965.

[34] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *International Conference on Rewriting Techniques and Applications (RTA2004).*

[35] Jacob Matthews and Robert Bruce Findler. An Operational Semantics for R5RS Scheme. In *2005 Workshop on Scheme and Functional Programming.* September 2005.

[36] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[37] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[38] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[39] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[40] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

[41] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[42] Jonathan Rees and William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[43] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[44] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[45] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[46] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[47] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[48] *Scheme Standardization charter.* `http://www.schemers.org/Documents/Standards/Charter/mar-2006.txt`, March 2006.

[49] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[50] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

[51] The Unicode Consortium. *The Unicode Standard, Version 5.0.0*, defined by: *The Unicode Standard, Version 5.0* (Boston, MA, Addison-Wesley, 2007. ISBN 0-321-48091-0).

[52] Oscar Waddell. *Extending the Scope of Syntactic Extension.* PhD thesis, Indiana University, August 1999.

[53] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.