Table of Contents

Chapter 1. Introduction ............................................................................................................. 9
  Section 1.1. Scheme Syntax ................................................................................................. 11
  Section 1.2. Scheme Naming Conventions ....................................................................... 12
  Section 1.3. Typographical and Notational Conventions .................................................. 13

Chapter 2. Getting Started .................................................................................................... 15
  Section 2.1. Interacting with Scheme ................................................................................. 15
  Section 2.2. Simple Expressions ....................................................................................... 17
  Section 2.3. Evaluating Scheme Expressions ................................................................... 22
  Section 2.4. Variables and Let Expressions ..................................................................... 23
  Section 2.5. Lambda Expressions ..................................................................................... 25
  Section 2.6. Top-Level Definitions .................................................................................... 29
  Section 2.7. Conditional Expressions .............................................................................. 32
  Section 2.8. Simple Recursion .......................................................................................... 37
  Section 2.9. Assignment ...................................................................................................... 42

Chapter 3. Going Further ....................................................................................................... 51
  Section 3.1. Syntactic Extension ....................................................................................... 51
  Exercise 3.1.1 .................................................................................................................... 54
  Section 3.2. More Recursion ............................................................................................. 55
  Section 3.3. Continuations ............................................................................................... 62
  Section 3.4. Continuation Passing Style ......................................................................... 66
  Section 3.5. Internal Definitions ...................................................................................... 68
  Section 3.6. Libraries ......................................................................................................... 72

Chapter 4. Procedures and Variable Bindings ...................................................................... 75
  Section 4.1. Variable References ...................................................................................... 75
  Section 4.2. Lambda ........................................................................................................... 75
  Section 4.3. Case-Lambda ................................................................................................. 76
  Section 4.4. Local Binding ................................................................................................. 78
  Section 4.5. Multiple Values ............................................................................................. 81
  Section 4.6. Variable Definitions ....................................................................................... 81
  Section 4.7. Assignment .................................................................................................... 83

Chapter 5. Control Operations .............................................................................................. 87
  Section 5.1. Procedure Application ................................................................................... 87
  Section 5.2. Sequencing ..................................................................................................... 88
  Section 5.3. Conditionals .................................................................................................. 88
  Section 5.4. Recursion and Iteration .............................................................................. 93
  Section 5.5. Mapping and Folding .................................................................................... 95
  Section 5.6. Continuations ............................................................................................... 99
  Section 5.7. Delayed Evaluation ..................................................................................... 103
  Section 5.8. Multiple Values ............................................................................................ 105
  Section 5.9. Eval .............................................................................................................. 110

Chapter 6. Operations on Objects ....................................................................................... 113
  Section 6.1. Constants and Quotation .............................................................................. 113
  Section 6.2. Generic Equivalence and Type Predicates ..................................................... 115
## Table of Contents

### Chapter 6. Operations on Objects
- Section 6.3. Lists and Pairs .......................................................... 124
- Section 6.4. Numbers ................................................................. 133
- Section 6.5. Fixnums ................................................................. 152
- Section 6.6. Flonums ................................................................. 160
- Section 6.7. Characters ............................................................ 167
- Section 6.8. Strings ................................................................. 170
- Section 6.9. Vectors ................................................................. 176
- Section 6.10. Bytevectors ......................................................... 179
- Section 6.11. Symbols .............................................................. 189
- Section 6.12. Booleans ............................................................ 191
- Section 6.13. Hashtables ......................................................... 191
- Section 6.14. Enumerations ..................................................... 196

### Chapter 7. Input and Output
- Section 7.1. Transcoders .......................................................... 202
- Section 7.2. Opening Files ....................................................... 203
- Section 7.3. Standard Ports ..................................................... 205
- Section 7.4. String and Bytevector Ports ................................. 206
- Section 7.5. Opening Custom Ports ......................................... 208
- Section 7.6. Port Operations .................................................... 210
- Section 7.7. Input Operations ................................................. 212
- Section 7.8. Output Operations .............................................. 216
- Section 7.9. Convenience I/O ................................................ 217
- Section 7.10. Filesystem Operations ........................................ 221
- Section 7.11. Bytevector/String Conversions ......................... 222

### Chapter 8. Syntactic Extension
- Section 8.1. Keyword Bindings .............................................. 225
- Section 8.2. Syntax-Rules Transformers ................................. 227
- Section 8.3. Syntax-Case Transformers .................................. 230
- Section 8.4. Examples ......................................................... 240

### Chapter 9. Records
- Section 9.1. Defining Records ............................................... 249
- Section 9.2. Procedural Interface .......................................... 255
- Section 9.3. Inspection .......................................................... 258

### Chapter 10. Libraries and Top-Level Programs
- Section 10.1. Standard Libraries ........................................... 263
- Section 10.2. Defining New Libraries ..................................... 264
- Section 10.3. Top-Level Programs ......................................... 268
- Section 10.4. Examples ......................................................... 269

### Chapter 11. Exceptions and Conditions
- Section 11.1. Raising and Handling Exceptions ..................... 273
- Section 11.2. Defining Condition Types ................................. 276
- Section 11.3. Standard Condition Types ............................... 279

### Chapter 12. Extended Examples
- Section 12.1. Matrix and Vector Multiplication .................... 289
- Exercise 12.1.1 ................................................................. 293
# Table of Contents

**Chapter 12. Extended Examples**

- Section 12.2. Sorting ..................................................................................................................... 293
- Section 12.3. A Set Constructor .................................................................................................... 295
- Section 12.4. Word Frequency Counting ...................................................................................... 298
- Section 12.5. Scheme Printer ........................................................................................................ 302
- Section 12.6. Formatted Output .................................................................................................... 305
- Section 12.7. A Meta-Circular Interpreter for Scheme ................................................................. 307
- Section 12.8. Defining Abstract Objects ....................................................................................... 311
- Section 12.9. Fast Fourier Transform ............................................................................................ 314
- Section 12.10. A Unification Algorithm ......................................................................................... 319
- Section 12.11. Multitasking with Engines ...................................................................................... 321

**References**........................................................................................................................................................... 329

**Answers to Selected Exercises**.......................................................................................................................... 333

**Formal Syntax**......................................................................................................................................................... 349

**Summary of Forms**............................................................................................................................................... 355

- The Scheme Programming Language, 4th Edition ............................................................................. 372
- Written by R. Kent Dybvig ......................................................................................................................... 372
- From the back cover ................................................................................................................................. 372

**Index**.................................................................................................................................................................... 375
Table of Contents

• Preface
• Chapter 1. Introduction
  ◆ Section 1.1. Scheme Syntax
  ◆ Section 1.2. Scheme Naming Conventions
  ◆ Section 1.3. Typographical and Notational Conventions
• Chapter 2. Getting Started
  ◆ Section 2.1. Interacting with Scheme
  ◆ Section 2.2. Simple Expressions
  ◆ Section 2.3. Evaluating Scheme Expressions
  ◆ Section 2.4. Variables and Let Expressions
  ◆ Section 2.5. Lambda Expressions
  ◆ Section 2.6. Top-Level Definitions
  ◆ Section 2.7. Conditional Expressions
  ◆ Section 2.8. Simple Recursion
  ◆ Section 2.9. Assignment
• Chapter 3. Going Further
  ◆ Section 3.1. Syntactic Extension
  ◆ Section 3.2. More Recursion
  ◆ Section 3.3. Continuations
  ◆ Section 3.4. Continuation Passing Style
  ◆ Section 3.5. Internal Definitions
  ◆ Section 3.6. Libraries
• Chapter 4. Procedures and Variable Bindings
  ◆ Section 4.1. Variable References
  ◆ Section 4.2. Lambda
  ◆ Section 4.3. Case-Lambda
  ◆ Section 4.4. Local Binding
  ◆ Section 4.5. Multiple Values
  ◆ Section 4.6. Variable Definitions
  ◆ Section 4.7. Assignment
• Chapter 5. Control Operations
  ◆ Section 5.1. Procedure Application
  ◆ Section 5.2. Sequencing
  ◆ Section 5.3. Conditionals
  ◆ Section 5.4. Recursion and Iteration
  ◆ Section 5.5. Mapping and Folding
  ◆ Section 5.6. Continuations
  ◆ Section 5.7. Delayed Evaluation
  ◆ Section 5.8. Multiple Values
  ◆ Section 5.9. Eval
• Chapter 6. Operations on Objects
  ◆ Section 6.1. Constants and Quotation
  ◆ Section 6.2. Generic Equivalence and Type Predicates
  ◆ Section 6.3. Lists and Pairs
  ◆ Section 6.4. Numbers
  ◆ Section 6.5. Fixnums
  ◆ Section 6.6. Flonums
  ◆ Section 6.7. Characters
  ◆ Section 6.8. Strings
  ◆ Section 6.9. Vectors
  ◆ Section 6.10. Bytevectors
  ◆ Section 6.11. Symbols
• Section 6.12. Booleans
• Section 6.13. Hashtables
• Section 6.14. Enumerations

• Chapter 7. Input and Output
  • Section 7.1. Transcoders
  • Section 7.2. Opening Files
  • Section 7.3. Standard Ports
  • Section 7.4. String and Bytevector Ports
  • Section 7.5. Opening Custom Ports
  • Section 7.6. Port Operations
  • Section 7.7. Input Operations
  • Section 7.8. Output Operations
  • Section 7.9. Convenience I/O
  • Section 7.10. Filesystem Operations
  • Section 7.11. Bytevector/String Conversions

• Chapter 8. Syntactic Extension
  • Section 8.1. Keyword Bindings
  • Section 8.2. Syntax-Rules Transformers
  • Section 8.3. Syntax-Case Transformers
  • Section 8.4. Examples

• Chapter 9. Records
  • Section 9.1. Defining Records
  • Section 9.2. Procedural Interface
  • Section 9.3. Inspection

• Chapter 10. Libraries and Top-Level Programs
  • Section 10.1. Standard Libraries
  • Section 10.2. Defining New Libraries
  • Section 10.3. Top-Level Programs
  • Section 10.4. Examples

• Chapter 11. Exceptions and Conditions
  • Section 11.1. Raising and Handling Exceptions
  • Section 11.2. Defining Condition Types
  • Section 11.3. Standard Condition Types

• Chapter 12. Extended Examples
  • Section 12.1. Matrix and Vector Multiplication
  • Section 12.2. Sorting
  • Section 12.3. A Set Constructor
  • Section 12.4. Word Frequency Counting
  • Section 12.5. Scheme Printer
  • Section 12.6. Formatted Output
  • Section 12.7. A Meta-Circular Interpreter for Scheme
  • Section 12.8. Defining Abstract Objects
  • Section 12.9. Fast Fourier Transform
  • Section 12.10. A Unification Algorithm
  • Section 12.11. Multitasking with Engines

• References
• Answers to Selected Exercises
• Formal Syntax
• Summary of Forms
• Index
Preface

Scheme was introduced in 1975 by Gerald J. Sussman and Guy L. Steele Jr. [28, 29], and was the first dialect of Lisp to fully support lexical scoping, first-class procedures, and continuations. In its earliest form it was a small language intended primarily for research and teaching, supporting only a handful of predefined syntactic forms and procedures. Scheme is now a complete general-purpose programming language, though it still derives its power from a small set of key concepts. Early implementations of the language were interpreter-based and slow, but some current Scheme implementations boast sophisticated compilers that generate code on par with code generated by the best optimizing compilers for lower-level languages such as C and Fortran.

This book is intended to provide an introduction to the Scheme programming language but not an introduction to programming in general. The reader is expected to have had some experience programming and to be familiar with terms commonly associated with computers and programming languages. Readers unfamiliar with Scheme or Lisp should also consider reading The Little Schemer [13] to become familiar with the concepts of list processing and recursion. Readers new to programming should begin with an introductory text on programming.

Scheme has been standardized both formally and informally. The IEEE Standard for the Scheme Programming Language [18], describes a formal ANSI/IEEE Standard for Scheme but dates back to 1991. A related series of reports, the "Revised Reports on the Algorithmic Language Scheme," document an evolving informal standard that most implementations support. The current report in this series is the “Revised6 Report on the Algorithmic Language Scheme” [24], which was completed in 2007.

This book covers the language of the Revised6 Report. It is not intended to supplant the Revised6 Report but rather to provide a more comprehensive introduction and reference manual for the language, with more explanatory text and examples, suitable more for users than for implementors. Features specific to particular implementations of Scheme are not included. In particular, features specific to the author’s Chez Scheme and Petite Chez Scheme implementations are described separately in the Chez Scheme User’s Guide [9]. On the other hand, no book on Scheme would be complete without some coverage of the interactive top level, since nearly every Scheme system supports interactive use in one form or another, even though the behavior is not standardized by the Revised6 Report. Chapters 2 and 3 are thus written assuming that the reader has available a Scheme implementation that supports an interactive top level, with behavior consistent with the description of the top-level environment in earlier reports and the IEEE/ANSI standard.

A large number of small- to medium-sized examples are spread throughout the text, and one entire chapter is dedicated to the presentation of a set of longer examples. Many of the examples show how a standard Scheme syntactic form or procedure might be implemented; others implement useful extensions. All of the examples can be entered directly from the keyboard into an interactive Scheme session.

This book is organized into twelve chapters, plus back matter. Chapter 1 describes the properties and features of Scheme that make it a useful and enjoyable language to use. Chapter 1 also describes Scheme’s notational conventions and the typographical conventions employed in this book.

Chapter 2 is an introduction to Scheme programming for the novice Scheme programmer that leads the reader through a series of examples, beginning with simple Scheme expressions and working toward progressively more difficult ones. Each section of Chapter 2 introduces a small set of related features, and the end of each section contains a set of exercises for further practice. The reader will learn the most from Chapter 2 by sitting at the keyboard and typing in the examples and trying the exercises.

Chapter 3 continues the introduction but covers more advanced features and concepts. Even readers with prior Scheme experience may wish to work through the examples and exercises found there.
Chapters 4 through 11 make up the reference portion of the text. They present each of Scheme's primitive procedures and syntactic forms in turn, grouping them into short sections of related procedures and forms. Chapter 4 describes operations for creating procedures and variable bindings; Chapter 5, program control operations; Chapter 6, operations on the various object types (including lists, numbers, and strings); Chapter 7, input and output operations; Chapter 8, syntactic extension; Chapter 9, record-type definitions; Chapter 10, libraries and top-level programs; and Chapter 11, exceptions and conditions.

Chapter 12 contains a collection of example procedures, libraries, and programs, each with a short overview, some examples of its use, the implementation with brief explanation, and a set of exercises for further work. Each of these programs demonstrates a particular set of features, and together they illustrate an appropriate style for programming in Scheme.

Following Chapter 12 are bibliographical references, answers to selected exercises, a detailed description of the formal syntax of Scheme programs and data, a concise summary of Scheme syntactic forms and procedures, and the index. The summary of forms and procedures is a useful first stop for programmers unsure of the structure of a syntactic form or the arguments expected by a primitive procedure. The page numbers appearing in the summary of forms and procedures and the italicized page numbers appearing in the index indicate the locations in the text where forms and procedures are defined.

Because the reference portion describes a number of aspects of the language not covered by the introductory chapters along with a number of interesting short examples, most readers will find it profitable to read through most of the material to become familiar with each feature and how it relates to other features. Chapter 6 is lengthy, however, and may be skimmed and later referenced as needed.

An online version of this book is available at http://www.scheme.com/tspl/. The summary of forms and index in the online edition include page numbers for the printed version and are thus useful as searchable indexes.

About the illustrations: The cover illustration and the illustration at the front of each chapter are algorithmic line fields created by artist Jean-Pierre Hébert, based on an idea inspired by the writings of John Cage. Each line field is created by the composition of any number of grids of parallel lines. The grids are regular, but they are not. For instance, the lines are of irregular length, which creates ragged edges. Their tone and thickness vary slightly. They are not exactly equidistant. They intersect with each other at a certain angle. When this angle is small, patterns of interference develop. The lines are first steeped into various scalar fields that perturb their original straight shape, then projected on the plane of the paper. Masks introduce holes in some layers. For the cover illustration, the grids are colored in different hues.

All the images are created by a single Scheme program that makes most of the decisions, based heavily on chance. The artist controls only canvas size, aspect ratio, the overall palette of colors, and levels of chance and fuzziness. The task of the artist is to introduce just enough chance at the right place so that the results are at the same time surprising, interesting, and in line with the artist's sense of aesthetics. This is a game of uncertainty, chaos, and harmony.

Acknowledgments: Many individuals contributed in one way or another to the preparation of one or more editions of this book, including Bruce Smith, Eugene Kohlbecker, Matthias Felleisen, Dan Friedman, Bruce Duba, Phil Dybvig, Guy Steele, Bob Hieb, Chris Haynes, Dave Plaisted, Joan Curry, Frank Silbermann, Pavel Curtis, John Wait, Carl Bruggeman, Sam Daniel, Oscar Waddell, Mike Ashley, John LaLonde, John Zuckerman, John Simmons, Bob Prior, Bob Burger, and Aziz Ghuloum. Many others have offered minor corrections and suggestions. Oscar Waddell helped create the typesetting system used to format the printed and online versions of this book. A small amount of text and a few examples have been adapted from the Revised6 Report for this book, for which credit goes to the editors of that report and many others who contributed to it. Finally and most importantly, my wife, Susan Dybvig, suggested that I write this book in the first place and lent her expertise and assistance to the production and publication of this and the previous editions.
Chapter 1. Introduction

Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters. While Scheme is often identified with symbolic applications, its rich set of data types and flexible control structures make it a truly versatile language. Scheme has been employed to write text editors, optimizing compilers, operating systems, graphics packages, expert systems, numerical applications, financial analysis packages, virtual reality systems, and practically every other type of application imaginable. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts and since the interactive nature of most implementations encourages experimentation. Scheme is a challenging language to understand fully, however; developing the ability to use its full potential requires careful study and practice.

Scheme programs are highly portable across versions of the same Scheme implementation on different machines, because machine dependencies are almost completely hidden from the programmer. They are also portable across different implementations because of the efforts of a group of Scheme language designers who have published a series of reports, the "Revised Reports" on Scheme. The most recent, the "Revised 6 Report" [24], emphasizes portability through a set of standard libraries and a standard mechanism for defining new portable libraries and top-level programs.

Although some early Scheme systems were inefficient and slow, many newer compiler-based implementations are fast, with programs running on par with equivalent programs written in lower-level languages. The relative inefficiency that sometimes remains results from run-time checks that support generic arithmetic and help programmers detect and correct various common programming errors. These checks may be disabled in many implementations.

Scheme supports many types of data values, or objects, including characters, strings, symbols, lists or vectors of objects, and a full set of numeric data types, including complex, real, and arbitrary-precision rational numbers.

The storage required to hold the contents of an object is dynamically allocated as necessary and retained until no longer needed, then automatically deallocated, typically by a garbage collector that periodically recovers the storage used by inaccessible objects. Simple atomic values, such as small integers, characters, booleans, and the empty list, are typically represented as immediate values and thus incur no allocation or deallocation overhead.

Regardless of representation, all objects are first-class data values; because they are retained indefinitely, they may be passed freely as arguments to procedures, returned as values from procedures, and combined to form new objects. This is in contrast with many other languages where composite data values such as arrays are either statically allocated and never deallocated, allocated on entry to a block of code and unconditionally deallocated on exit from the block, or explicitly allocated and deallocated by the programmer.

Scheme is a call-by-value language, but for at least mutable objects (objects that can be modified), the values are pointers to the actual storage. These pointers remain behind the scenes, however, and programmers need not be conscious of them except to understand that the storage for an object is not copied when an object is passed to or returned from a procedure.

At the heart of the Scheme language is a small core of syntactic forms from which all other forms are built. These core forms, a set of extended syntactic forms derived from them, and a set of primitive procedures make up the full Scheme language. An interpreter or compiler for Scheme can be quite small and potentially fast and highly reliable. The extended syntactic forms and many primitive procedures can be defined in Scheme itself, simplifying the implementation and increasing reliability.
Scheme programs share a common printed representation with Scheme data structures. As a result, any Scheme program has a natural and obvious internal representation as a Scheme object. For example, variables and syntactic keywords correspond to symbols, while structured syntactic forms correspond to lists. This representation is the basis for the syntactic extension facilities provided by Scheme for the definition of new syntactic forms in terms of existing syntactic forms and procedures. It also facilitates the implementation of interpreters, compilers, and other program transformation tools for Scheme directly in Scheme, as well as program transformation tools for other languages in Scheme.

Scheme variables and keywords are lexically scoped, and Scheme programs are block-structured. Identifiers may be imported into a program or library or bound locally within a given block of code such as a library, program, or procedure body. A local binding is visible only lexically, i.e., within the program text that makes up the particular block of code. An occurrence of an identifier of the same name outside this block refers to a different binding; if no binding for the identifier exists outside the block, then the reference is invalid. Blocks may be nested, and a binding in one block may shadow a binding for an identifier of the same name in a surrounding block. The scope of a binding is the block in which the bound identifier is visible minus any portions of the block in which the identifier is shadowed. Block structure and lexical scoping help create programs that are modular, easy to read, easy to maintain, and reliable. Efficient code for lexical scoping is possible because a compiler can determine before program evaluation the scope of all bindings and the binding to which each identifier reference resolves. This does not mean, of course, that a compiler can determine the values of all variables, since the actual values are not computed in most cases until the program executes.

In most languages, a procedure definition is simply the association of a name with a block of code. Certain variables local to the block are the parameters of the procedure. In some languages, a procedure definition may appear within another block or procedure so long as the procedure is invoked only during execution of the enclosing block. In others, procedures can be defined only at top level. In Scheme, a procedure definition may appear within another block or procedure, and the procedure may be invoked at any time thereafter, even if the enclosing block has completed its execution. To support lexical scoping, a procedure carries the lexical context (environment) along with its code.

Furthermore, Scheme procedures are not always named. Instead, procedures are first-class data objects like strings or numbers, and variables are bound to procedures in the same way they are bound to other objects.

As with procedures in most other languages, Scheme procedures may be recursive. That is, any procedure may invoke itself directly or indirectly. Many algorithms are most elegantly or efficiently specified recursively. A special case of recursion, called tail recursion, is used to express iteration, or looping. A tail call occurs when one procedure directly returns the result of invoking another procedure; tail recursion occurs when a procedure recursively tail-calls itself, directly or indirectly. Scheme implementations are required to implement tail calls as jumps (gotos), so the storage overhead normally associated with recursion is avoided. As a result, Scheme programmers need master only simple procedure calls and recursion and need not be burdened with the usual assortment of looping constructs.

Scheme supports the definition of arbitrary control structures with continuations. A continuation is a procedure that embodies the remainder of a program at a given point in the program. A continuation may be obtained at any time during the execution of a program. As with other procedures, a continuation is a first-class object and may be invoked at any time after its creation. Whenever it is invoked, the program immediately continues from the point where the continuation was obtained. Continuations allow the implementation of complex control mechanisms including explicit backtracking, multithreading, and coroutines.

Scheme also allows programmers to define new syntactic forms, or syntactic extensions, by writing transformation procedures that determine how each new syntactic form maps to existing syntactic forms. These transformation procedures are themselves expressed in Scheme with the help of a convenient high-level pattern language that automates syntax checking, input deconstruction, and output reconstruction. By default,
lexical scoping is maintained through the transformation process, but the programmer can exercise control over the scope of all identifiers appearing in the output of a transformer. Syntactic extensions are useful for defining new language constructs, for emulating language constructs found in other languages, for achieving the effects of in-line code expansion, and even for emulating entire languages in Scheme. Most large Scheme programs are built from a mix of syntactic extensions and procedure definitions.

Scheme evolved from the Lisp language and is considered to be a dialect of Lisp. Scheme inherited from Lisp the treatment of values as first-class objects, several important data types, including symbols and lists, and the representation of programs as objects, among other things. Lexical scoping and block structure are features taken from Algol 60 [21]. Scheme was the first Lisp dialect to adopt lexical scoping and block structure, first-class procedures, the treatment of tail calls as jumps, continuations, and lexically scoped syntactic extensions.

Common Lisp [27] and Scheme are both contemporary Lisp languages, and the development of each has been influenced by the other. Like Scheme but unlike earlier Lisp languages, Common Lisp adopted lexical scoping and first-class procedures, although Common Lisp's syntactic extension facility does not respect lexical scoping. Common Lisp's evaluation rules for procedures are different from the evaluation rules for other objects, however, and it maintains a separate namespace for procedure variables, thereby inhibiting the use of procedures as first-class objects. Also, Common Lisp does not support continuations or proper treatment of tail calls, but it does support several less general control structures not found in Scheme. While the two languages are similar, Common Lisp includes more specialized constructs, while Scheme includes more general-purpose building blocks out of which such constructs (and others) may be built.

The remainder of this chapter describes Scheme's syntax and naming conventions and the typographical conventions used throughout this book.

Section 1.1. Scheme Syntax

Scheme programs are made up of keywords, variables, structured forms, constant data (numbers, characters, strings, quoted vectors, quoted lists, quoted symbols, etc.), whitespace, and comments.

Keywords, variables, and symbols are collectively called identifiers. Identifiers may be formed from letters, digits, and certain special characters, including ?, !, +, -, *, /, <, >, ::, $, %, ^, &, _, ~, and @, as well as a set of additional Unicode characters. Identifiers cannot start with an at sign (@) and normally cannot start with any character that can start a number, i.e., a digit, plus sign (+), minus sign (-), or decimal point (.). Exceptions are +, -, and . . . , which are valid identifiers, and any identifier starting with ->. For example, hi, Hello, n, x, x3, x+2, and ?$&*!!! are all identifiers. Identifiers are delimited by whitespace, comments, parentheses, brackets, string (double) quotes ("), and hash marks (#). A delimiter or any other Unicode character may be included anywhere within the name of an identifier as an escape of the form \xsv; where sv is the scalar value of the character in hexadecimal notation.

There is no inherent limit on the length of a Scheme identifier; programmers may use as many characters as necessary. Long identifiers are no substitute for comments, however, and frequent use of long identifiers can make a program difficult to format and consequently difficult to read. A good rule is to use short identifiers when the scope of the identifier is small and longer identifiers when the scope is larger.

Identifiers may be written in any mix of upper- and lower-case letters, and case is significant, i.e., two identifiers are different even if they differ only in case. For example, abcdE, Abcede, AbCdE, and ABCDE all refer to different identifiers. This is a change from previous versions of the Revised Report.

Structured forms and list constants are enclosed within parentheses, e.g., (a b c) or (* (- x 2) y). The empty list is written (). Matched sets of brackets ([ ]) may be used in place of parentheses and are often used to set off the subexpressions of certain standard syntactic forms for readability, as shown in examples throughout this book. Vectors are written similarly to lists, except that they are preceded by # (and
terminated by), e.g., #(this is a vector of symbols). Bytevectors are written as sequences of unsigned byte values (exact integers in the range 0 through 255) bracketed by #vu8 (and), e.g., #vu8(3 250 45 73).

Strings are enclosed in double quotation marks, e.g., "I am a string". Characters are preceded by #\, e.g., #\a. Case is important within character and string constants, as within identifiers. Numbers may be written as integers, e.g., -123, as ratios, e.g., 1/2, in floating-point or scientific notation, e.g., 1.3 or 1e23, or as complex numbers in rectangular or polar notation, e.g., 1.3-2.7i or -1.2@73. Case is not important in the syntax of a number. The boolean values representing true and false are written #t and #f. Scheme conditional expressions actually treat #f as false and all other objects as true, so 3.0, (), "false", and nil all count as true.

Details of the syntax for each type of constant data are given in the individual sections of Chapter 6 and in the formal syntax of Scheme starting on page 455.

Scheme expressions may span several lines, and no explicit terminator is required. Since the number of whitespace characters (spaces and newlines) between expressions is not significant, Scheme programs should be indented to show the structure of the code in a way that makes the code as readable as possible. Comments may appear on any line of a Scheme program, between a semicolon (;) and the end of the line. Comments explaining a particular Scheme expression are normally placed at the same indentation level as the expression, on the line before the expression. Comments explaining a procedure or group of procedures are normally placed before the procedures, without indentation. Multiple comment characters are often used to set off the latter kind of comment, e.g., ;; The following procedures ....

Two other forms of comments are supported: block comments and datum comments. Block comments are delimited by #| and |# pairs, and may be nested. A datum comment consists of a # prefix and the datum (printed data value) that follows it. Datum comments are typically used to comment out individual definitions or expressions. For example, (three #; (not four) element list) is just what it says. Datum comments may also be nested, though #; #; (a) (b) has the somewhat nonobvious effect of commenting out both (a) and (b).

Some Scheme values, such as procedures and ports, do not have standard printed representations and can thus never appear as a constant in the printed syntax of a program. This book uses the notation #<description> when showing the output of an operation that returns such a value, e.g., #<procedure> or #<port>.

**Section 1.2. Scheme Naming Conventions**

Scheme’s naming conventions are designed to provide a high degree of regularity. The following is a list of these naming conventions:

- Predicate names end in a question mark (?). Predicates are procedures that return a true or false answer, such as eq?, zero?, and string=? . The common numeric comparators =, <, >, <=, and => are exceptions to this naming convention.
- Type predicates, such as pair?, are created from the name of the type, in this case pair, and the question mark.
- The names of most character, string, and vector procedures start with the prefix char-, string-, and vector-, e.g., string-append. (The names of some list procedures start with list-, but most do not.)
- The names of procedures that convert an object of one type into an object of another type are written as type1->type2, e.g., vector->list.
- The names of procedures and syntactic forms that cause side effects end with an exclamation point (!). These include set! and vector-set!. Procedures that perform input or output technically cause side effects, but their names are exceptions to this rule.
Programmers should employ these same conventions in their own code whenever possible.

**Section 1.3. Typographical and Notational Conventions**

A standard procedure or syntactic form whose sole purpose is to perform some side effect is said to return *unspecified*. This means that an implementation is free to return any number of values, each of which can be any Scheme object, as the value of the procedure or syntactic form. Do not count on these values being the same across implementations, the same across versions of the same implementation, or even the same across two uses of the procedure or syntactic form. Some Scheme systems routinely use a special object to represent unspecified values. Printing of this object is often suppressed by interactive Scheme systems, so that the values of expressions returning unspecified values are not printed.

While most standard procedures return a single value, the language supports procedures that return zero, one, more than one, or even a variable number of values via the mechanisms described in Section 5.8. Some standard expressions can evaluate to multiple values if one of their subexpressions evaluates to multiple values, e.g., by calling a procedure that returns multiple values. When this situation can occur, an expression is said to return "the values" rather than simply "the value" of its subexpression. Similarly, a standard procedure that returns the values resulting from a call to a procedure argument is said to return the values returned by the procedure argument.

This book uses the words "must" and "should" to describe program requirements, such as the requirement to provide an index that is less than the length of the vector in a call to `vector-ref`. If the word "must" is used, it means that the requirement is enforced by the implementation, i.e., an exception is raised, usually with condition type `&assertion`. If the word "should" is used, an exception may or may not be raised, and if not, the behavior of the program is undefined.

The phrase "syntax violation" is used to describe a situation in which a program is malformed. Syntax violations are detected prior to program execution. When a syntax violation is detected, an exception of type `&syntax` is raised and the program is not executed.

The typographical conventions used in this book are straightforward. All Scheme objects are printed in a typewriter typeface, just as they are to be typed at the keyboard. This includes syntactic keywords, variables, constant objects, Scheme expressions, and example programs. An italic typeface is used to set off syntax variables in the descriptions of syntactic forms and arguments in the descriptions of procedures. Italics are also used to set off technical terms the first time they appear. In general, names of syntactic forms and procedures are never capitalized, even at the beginning of a sentence. The same is true for syntax variables written in italics.

In the description of a syntactic form or procedure, one or more prototype patterns show the syntactic form or forms or the correct number or numbers of arguments for an application of the procedure. The keyword or procedure name is given in typewriter font, as are parentheses. The remaining pieces of the syntax or arguments are shown in italics, using a name that implies the type of expression or argument expected by the syntactic form or procedure. Ellipses are used to specify zero or more occurrences of a subexpression or argument. For example, `(or expr ...)` describes the `or` syntactic form, which has zero or more subexpressions, and `(member obj list)` describes the `member` procedure, which expects two arguments, an object and a list.

A syntax violation occurs if the structure of a syntactic form does not match its prototype. Similarly, an exception with condition type `&assertion` is raised if the number of arguments passed to a standard procedure does not match what it is specified to receive. An exception with condition type `&assertion` is also raised if a standard procedure receives an argument whose type is not the type implied by its name or does not meet other criteria given in the description of the procedure. For example, the prototype for `vector-set!` is
(vector-set! vector n obj)

and the description says that $n$ must be an exact nonnegative integer strictly less than the length of vector. Thus, vector-set! must receive three arguments, the first of which must be a vector, the second of which must be an exact nonnegative integer less than the length of the vector, and the third of which may be any Scheme value. Otherwise, an exception with condition type &assertion is raised.

In most cases, the type of argument required is obvious, as with vector, obj, or binary-input-port. In others, primarily within the descriptions of numeric routines, abbreviations are used, such as int for integer, exint for exact integer, and fx for fixnum. These abbreviations are explained at the start of the sections containing the affected entries.
Chapter 2. Getting Started

This chapter is an introduction to Scheme for programmers who are new to the language. You will get more from this chapter if you are sitting in front of an interactive Scheme system, trying out the examples as you go.

After reading this chapter and working the exercises, you should be able to start using Scheme. You will have learned the syntax of Scheme programs and how they are executed, along with how to use simple data structures and control mechanisms.

Section 2.1. Interacting with Scheme

Most Scheme systems provide an interactive programming environment that simplifies program development and experimentation. The simplest interaction with Scheme follows a "read-evaluate-print" cycle. A program (often called a read-evaluate-print loop, or REPL) reads each expression you type at the keyboard, evaluates it, and prints its value.

With an interactive Scheme system, you can type an expression at the keyboard and see its value immediately. You can define a procedure and apply it to arguments to see how it works. You can even type in an entire program consisting of a set of procedure definitions and test it without leaving the system. When your program starts getting longer, it will be more convenient to type it into a file (using a text editor), load the file and test it interactively. In most Scheme systems, a file may be loaded with the nonstandard procedure load, which takes a string argument naming the file. Preparing your program in a file has several advantages: you have a chance to compose your program more carefully, you can correct errors without retyping the program, and you can retain a copy for later use. Most Scheme implementations treat expressions loaded from a file the same as expressions typed at the keyboard.

While Scheme provides various input and output procedures, the REPL takes care of reading expressions and printing their values. This frees you to concentrate on writing your program without worrying about how its results will be displayed.

The examples in this chapter and in the rest of the book follow a regular format. An expression you might type from your keyboard is given first, possibly spanning several lines. The value of the expression is given after the to be read as "evaluates to." The is omitted for definitions and when the value of an expression is unspecified.

The example programs are formatted in a style that "looks nice" and conveys the structure of the program. The code is easy to read because the relationship between each expression and its subexpressions is clearly shown. Scheme ignores indentation and line breaks, however, so there is no need to follow a particular style. The important thing is to establish one style and keep to it. Scheme sees each program as if it were on a single line, with its subexpressions ordered from left to right.

If you have access to an interactive Scheme system, it might be a good idea to start it up now and type in the examples as you read. One of the simplest Scheme expressions is a string constant. Try typing "Hi Mom!" (including the double quotes) in response to the prompt. The system should respond with "Hi Mom!"; the value of any constant is the constant itself.

"Hi Mom!" \Rightarrow "Hi Mom!"

Here is a set of expressions, each with Scheme's response. They are explained in later sections of this chapter, but for now use them to practice interacting with Scheme.

"hello" \Rightarrow "hello"
42 \Rightarrow 42
22/7 \Rightarrow 22/7
3.141592653 \Rightarrow 3.141592653
+ \Rightarrow \#<procedure>
(+ 76 31) \Rightarrow 107
(* -12 10) \Rightarrow -120
'(a b c d) \Rightarrow (a b c d)

Be careful not to miss any single quotes ('), double quotes, or parentheses. If you left off a single quote in
the last expression, you probably received a message indicating that an exception has occurred. Just try again.
If you left off a closing parenthesis or double quote, the system might still be waiting for it.

Here are a few more expressions to try. You can try to figure out on your own what they mean or wait to find
out later in the chapter.

(car '(a b c)) \Rightarrow a
(cdr '(a b c)) \Rightarrow (b c)
(cons 'a '(b c)) \Rightarrow (a b c)
(cons (car '(a b c)) (cdr '(d e f))) \Rightarrow (a e f)

As you can see, Scheme expressions may span more than one line. The Scheme system knows when it has an
entire expression by matching double quotes and parentheses.

Next, let's try defining a procedure.

(define square
  (lambda (n)
    (* n n)))

The procedure \texttt{square} computes the square $n^2$ of any number $n$. We say more about the expressions that
make up this definition later in this chapter. For now it suffices to say that \texttt{define} establishes variable
bindings, \texttt{lambda} creates procedures, and * names the multiplication procedure. Note the form of these
expressions. All structured forms are enclosed in parentheses and written in \textit{prefix notation}, i.e., the operator
precedes the arguments. As you can see, this is true even for simple arithmetic operations such as *.

Try using \texttt{square}.

(square 5) \Rightarrow 25
(square -200) \Rightarrow 40000
(square 0.5) \Rightarrow 0.25
(square -1/2) \Rightarrow 1/4

Even though the next definition is short, you might enter it into a file. Let's assume you call the file
"reciprocal.ss."

(define reciprocal
  (lambda (n)
    (if (= n 0)
        "oops!"
        (/ 1 n))))

This procedure, \texttt{reciprocal}, computes the quantity $1/n$ for any number $n \neq 0$. For $n = 0$, \texttt{reciprocal}
returns the string "oops!": Return to Scheme and try loading your file with the procedure \texttt{load}.
Finally, try using the procedure we have just defined.

(reciprocal 10) ⇒ 1/10
(reciprocal 1/10) ⇒ 10
(reciprocal 0) ⇒ "oops!"
(reciprocal (reciprocal 1/10)) ⇒ 1/10

In the next section we will discuss Scheme expressions in more detail. Throughout this chapter, keep in mind that your Scheme system is one of the most useful tools for learning Scheme. Whenever you try one of the examples in the text, follow it up with your own examples. In an interactive Scheme system, the cost of trying something out is relatively small—usually just the time to type it in.

Section 2.2. Simple Expressions

The simplest Scheme expressions are constant data objects, such as strings, numbers, symbols, and lists. Scheme supports other object types, but these four are enough for many programs. We saw some examples of strings and numbers in the preceding section.

Let's discuss numbers in a little more detail. Numbers are constants. If you enter a number, Scheme echoes it back to you. The following examples show that Scheme supports several types of numbers.

123456789987654321 ⇒ 123456789987654321
3/4 ⇒ 3/4
2.718281828 ⇒ 2.718281828
2.2+1.1i ⇒ 2.2+1.1i

Scheme numbers include exact and inexact integer, rational, real, and complex numbers. Exact integers and rational numbers have arbitrary precision, i.e., they can be of arbitrary size. Inexact numbers are usually represented internally using IEEE standard floating-point representations.

Scheme provides the names +, -, *, and / for the corresponding arithmetic procedures. Each procedure accepts two numeric arguments. The expressions below are called procedure applications, because they specify the application of a procedure to a set of arguments.

(+ 1/2 1/2) ⇒ 1
(- 1.5 1/2) ⇒ 1.0

(* 3 1/2) ⇒ 3/2
(/ 1.5 3/4) ⇒ 2.0

Scheme employs prefix notation even for common arithmetic operations. Any procedure application, whether the procedure takes zero, one, two, or more arguments, is written as (procedure arg ...). This regularity simplifies the syntax of expressions; one notation is employed regardless of the operation, and there are no complicated rules regarding the precedence or associativity of operators.

Procedure applications may be nested, in which case the innermost values are computed first. We can thus nest applications of the arithmetic procedures given above to evaluate more complicated formulas.

(+ (+ 2 2) (+ 2 2)) ⇒ 8
(- 2 (* 4 1/3)) ⇒ 2/3
(* 2 (* 2 (* 2 (* 2 2)))) ⇒ 32
(/ (* 6/7 7/2) (- 4.5 1.5)) ⇒ 1.0
These examples demonstrate everything you need to use Scheme as a four-function desk calculator. While we will not discuss them in this chapter, Scheme supports many other arithmetic procedures. Now might be a good time to turn to Section 6.4 and experiment with some of them.

Simple numeric objects are sufficient for many tasks, but sometimes aggregate data structures containing two or more values are needed. In many languages, the basic aggregate data structure is the array. In Scheme, it is the list. Lists are written as sequences of objects surrounded by parentheses. For instance, \((1 \ 2 \ 3 \ 4 \ 5)\) is a list of numbers, and \("\text{this} \ \text{is} \ \text{a} \ \text{list}\)\) is a list of strings. Lists need not contain only one type of object, so \((4.2 \ \text{"hi"})\) is a valid list containing a number and a string. Lists may be nested (may contain other lists), so \(((1\ 2) \ (3\ 4))\) is a valid list with two elements, each of which is a list of two elements.

You might notice that lists look just like procedure applications and wonder how Scheme tells them apart. That is, how does Scheme distinguish between a list of objects, \((\text{obj}_1 \ \text{obj}_2 \ \ldots)\), and a procedure application, \((\text{procedure} \ \text{arg} \ \ldots)\)?

In some cases, the distinction might seem obvious. The list of numbers \((1 \ 2 \ 3 \ 4 \ 5)\) could hardly be confused with a procedure application, since 1 is a number, not a procedure. So, the answer might be that Scheme looks at the first element of the list or procedure application and makes its decision based on whether that first element is a procedure or not. This answer is not good enough, since we might even want to treat a valid procedure application such as \((+ \ 3 \ 4)\) as a list. The answer is that we must tell Scheme explicitly to treat a list as data rather than as a procedure application. We do this with \(\text{quote}\).

\begin{verbatim}
(quote (1 2 3 4 5)) \Rightarrow (1 2 3 4 5)
(quote ("this" "is" "a" "list")) \Rightarrow ("this" "is" "a" "list")
(quote (+ 3 4)) \Rightarrow (+ 3 4)
\end{verbatim}

The \(\text{quote}\) forces the list to be treated as data. Try entering the above expressions without the quote; you will likely receive a message indicating that an exception has occurred for the first two and an incorrect answer (7) for the third.

Because \(\text{quote}\) is required fairly frequently in Scheme code, Scheme recognizes a single quotation mark (\(\text{'}\)) preceding an expression as an abbreviation for \(\text{quote}\).

\begin{verbatim}
'(1 2 3 4) \Rightarrow (1 2 3 4)
'((1 2) (3 4)) \Rightarrow ((1 2) (3 4))
'(/ (* 2 -1) 3) \Rightarrow (/ (* 2 -1) 3)
\end{verbatim}

Both forms are referred to as \(\text{quote}\) expressions. We often say an object is \(\text{quoted}\) when it is enclosed in a \(\text{quote}\) expression.

A \(\text{quote}\) expression is \textit{not} a procedure application, since it inhibits the evaluation of its subexpression. It is an entirely different syntactic form. Scheme supports several other syntactic forms in addition to procedure applications and \(\text{quote}\) expressions. Each syntactic form is evaluated differently. Fortunately, the number of different syntactic forms is small. We will see more of them later in this chapter.

Not all \(\text{quote}\) expressions involve lists. Try the following expression with and without the \(\text{quote}\) wrapper.

\begin{verbatim}
(quote hello) \Rightarrow hello
\end{verbatim}

The symbol \(\text{hello}\) must be quoted in order to prevent Scheme from treating \(\text{hello}\) as a \textit{variable}. Symbols and variables in Scheme are similar to symbols and variables in mathematical expressions and equations. When we evaluate the mathematical expression \(1 - x\) for some value of \(x\), we think of \(x\) as a variable. On the other hand, when we consider the algebraic equation \(x^2 - 1 = (x - 1)(x + 1)\), we think of \(x\) as a symbol (in fact,
we think of the whole equation symbolically). Just as quoting a list tells Scheme to treat a parenthesized form as a list rather than as a procedure application, quoting an identifier tells Scheme to treat the identifier as a symbol rather than as a variable. While symbols are commonly used to represent variables in symbolic representations of equations or programs, symbols may also be used, for example, as words in the representation of natural language sentences.

You might wonder why applications and variables share notations with lists and symbols. The shared notation allows Scheme programs to be represented as Scheme data, simplifying the writing of interpreters, compilers, editors, and other tools in Scheme. This is demonstrated by the Scheme interpreter given in Section 12.7, which is itself written in Scheme. Many people believe this to be one of the most important features of Scheme.

Numbers and strings may be quoted, too.

'2 ⇒ 2
'2/3 ⇒ 2/3
(quote "Hi Mom!") ⇒ "Hi Mom!"

Numbers and strings are treated as constants in any case, however, so quoting them is unnecessary.

Now let's discuss some Scheme procedures for manipulating lists. There are two basic procedures for taking lists apart: car and cdr (pronounced could-er). car returns the first element of a list, and cdr returns the remainder of the list. (The names "car" and "cdr" are derived from operations supported by the first computer on which a Lisp language was implemented, the IBM 704.) Each requires a nonempty list as its argument.

(car '(a b c)) ⇒ a
(cdr '(a b c)) ⇒ (b c)
(cdr '(a)) ⇒ ()
(cdr (car '(a b))) ⇒ b
(cdr (cdr '(a b c))) ⇒ (c)

(car '((a b) (c d))) ⇒ (a b)
(cdr '((a b) (c d))) ⇒ ((c d))

The first element of a list is often called the "car" of the list, and the rest of the list is often called the "cdr" of the list. The cdr of a list with one element is (), the empty list.

The procedure cons constructs lists. It takes two arguments. The second argument is usually a list, and in that case cons returns a list.

(cons 'a '()) ⇒ (a)
(cons 'a '(b c)) ⇒ (a b c)
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
(cons '(a b) 'c (d e f)) ⇒ ((a b) c d)

(car (cons 'a '(b c))) ⇒ a
(cdr (cons 'a '(b c))) ⇒ (b c)
(cdr (cons 'a (cons 'b (cons 'c '())))) ⇒ (a b c)
(car (cons 'a (cons 'b (cons 'c '())))) ⇒ ((a b) c)

car (cons 'a (cons 'b (cons 'c '()))) ⇒ a
(cdr (cons 'a (cons 'b (cons 'c '())))) ⇒ (b c)

Just as "car" and "cdr" are often used as nouns, "cons" is often used as a verb. Creating a new list by adding an
element to the beginning of a list is referred to as **consing** the element onto the list.

Notice the word "usually" in the description of cons's second argument. The procedure `cons` actually builds pairs, and there is no reason that the cdr of a pair must be a list. A list is a sequence of pairs; each pair's cdr is the next pair in the sequence.

```
| a |  |  | b |  |  | c |  | c | () |
```

The cdr of the last pair in a *proper list* is the empty list. Otherwise, the sequence of pairs forms an *improper list*. More formally, the empty list is a proper list, and any pair whose cdr is a proper list is a proper list.

An improper list is printed in **dotted-pair notation**, with a period, or *dot*, preceding the final element of the list.

```
(cons 'a 'b) ⇒ (a . b)
(cdr '(a . b)) ⇒ b
(cons 'a '(b . c)) ⇒ (a b . c)
```

Because of its printed notation, a pair whose cdr is not a list is often called a *dotted pair*. Even pairs whose cdrs are lists can be written in dotted-pair notation, however, although the printer always chooses to write proper lists without dots.

```
'(a . (b . (c . ()))) ⇒ (a b c)
```

The procedure `list` is similar to `cons`, except that it takes an arbitrary number of arguments and always builds a proper list.

```
(list 'a 'b 'c) ⇒ (a b c)
(list 'a) ⇒ (a)
(list) ⇒ ()
```

Section 6.3 provides more information on lists and the Scheme procedures for manipulating them. This might be a good time to turn to that section and familiarize yourself with the other procedures given there.

**Exercise 2.2.1**

Convert the following arithmetic expressions into Scheme expressions and evaluate them.

```
a. 1.2 × (2 - 1/3) + -8.7
b. (2/3 + 4/9) ÷ (5/11 - 4/3)
c. 1 ÷ (2 + 1 ÷ (1 + 1/2))
d. 1 × -2 × 3 × -4 × 5 × -6 × 7
```

**Exercise 2.2.2**

Experiment with the procedures `+`, `−`, `*`, and `/` to determine Scheme's rules for the type of value returned by each when given different types of numeric arguments.

**Exercise 2.2.3**

Determine the values of the following expressions. Use your Scheme system to verify your answers.

```
a. (cons 'car 'cdr)
```
b. (list 'this '(is silly))
c. (cons 'is '(this silly?))
d. (quote (+ 2 3))
e. (cons '+ '(2 3))
f. (car '+ (2 3))
g. (cdr '+ (2 3))
h. cons
i. (quote cons)
j. (quote (quote cons))
k. (car (quote (quote cons)))
l. (+ 2 3)
m. (+ '2 '3)
n. (+ (car '(2 3)) (car (cdr '(2 3))))
o. ((car (list + * /)) 2 3)

Exercise 2.2.4

(car (car '(((a b) (c d))))) yields a. Determine which compositions of car and cdr applied to ((a b) (c d)) yield b, c, and d.

Exercise 2.2.5

Write a Scheme expression that evaluates to the following internal list structure.

Exercise 2.2.6

Draw the internal list structure produced by the expression below.

(cons 1 (cons '(2 . ((3) . ())) (cons '() (cons 4 5))))

Exercise 2.2.7

The behavior of (car (car (car '(((a b) (c d))))) is undefined because (car '(((a b) (c d))) is (a b), (car '((a b))) is a, and (car 'a) is undefined. Determine all legal compositions of car and cdr applied to ((a b) (c d)).

Exercise 2.2.8

Try to explain how Scheme expressions are evaluated. Does your explanation cover the last example in Exercise 2.2.3?
Section 2.3. Evaluating Scheme Expressions

Let's turn to a discussion of how Scheme evaluates the expressions you type. We have already established the rules for constant objects such as strings and numbers: the object itself is the value. You have probably also worked out in your mind a rule for evaluating procedure applications of the form

\[(\text{procedure } \text{arg}_1 \ldots \text{arg}_n)\].

Here, procedure is an expression representing a Scheme procedure, and arg\_1 \ldots arg\_n are expressions representing its arguments. One possibility is the following.

- Find the value of procedure.
- Find the value of arg\_1.
  - : Find the value of arg\_n.
- Apply the value of procedure to the values of arg\_1 \ldots arg\_n.

For example, consider the simple procedure application \((+ 3 4)\). The value of + is the addition procedure, the value of 3 is the number 3, and the value of 4 is the number 4. Applying the addition procedure to 3 and 4 yields 7, so our value is the object 7.

By applying this process at each level, we can find the value of the nested expression \((* (+ 3 4) 2)\). The value of * is the multiplication procedure, the value of \((+ 3 4)\) we can determine to be the number 7, and the value of 2 is the number 2. Multiplying 7 by 2 we get 14, so our answer is 14.

This rule works for procedure applications but not for quote expressions because the subexpressions of a procedure application are evaluated, whereas the subexpression of a quote expression is not. The evaluation of a quote expression is more similar to the evaluation of constant objects. The value of a quote expression of the form \((\text{quote object})\) is simply object.

Constant objects, procedure applications, and quote expressions are only three of the many syntactic forms provided by Scheme. Fortunately, only a few of the other syntactic forms need to be understood directly by a Scheme programmer; these are referred to as core syntactic forms. The remaining syntactic forms are syntactic extensions defined, ultimately, in terms of the core syntactic forms. We will discuss the remaining core syntactic forms and a few syntactic extensions in the remaining sections of this chapter. Section 3.1 summarizes the core syntactic forms and introduces the syntactic extension mechanism.

Before we go on to more syntactic forms and procedures, two points related to the evaluation of procedure applications are worthy of note. First, the process given above is overspecified, in that it requires the subexpressions to be evaluated from left to right. That is, procedure is evaluated before arg\_1, arg\_1 is evaluated before arg\_2, and so on. This need not be the case. A Scheme evaluator is free to evaluate the expressions in any order---left to right, right to left, or any other sequential order. In fact, the subexpressions may be evaluated in different orders for different applications, even in the same implementation.

The second point is that procedure is evaluated in the same way as arg\_1 \ldots arg\_n. While procedure is often a variable that names a particular procedure, this need not be the case. Exercise 2.2.3 had you determine the value of the expression \(((\text{car (list + - * /)}) 2 3)\). Here, procedure is \((\text{car (list + - * /)})\). The value of \((\text{car (list + - * /)})\) is the addition procedure, just as if procedure were simply the variable +.

Exercise 2.3.1

Write down the steps necessary to evaluate the expression below.

\(((\text{car (cdr (list + - * /)}) 17 5)\)
Section 2.4. Variables and Let Expressions

Suppose $expr$ is a Scheme expression that contains a variable $var$. Suppose, additionally, that we would like $var$ to have the value $val$ when we evaluate $expr$. For example, we might like $x$ to have the value 2 when we evaluate $(+ x 3)$. Or, we might want $y$ to have the value 3 when we evaluate $(+ 2 y)$. The following examples demonstrate how to do this using Scheme's let syntactic form.

```
(let ((x 2))
  (+ x 3)) ⇒ 5

(let ((y 3))
  (+ 2 y)) ⇒ 5

(let ((x 2) (y 3))
  (+ x y)) ⇒ 5
```

The let syntactic form includes a list of variable-expression pairs, along with a sequence of expressions referred to as the body of the let. The general form of a let expression is

```
(let ((var expr) ...) body_1 body_2 ...)
```

We say the variables are bound to the values by the let. We refer to variables bound by let as let-bound variables.

A let expression is often used to simplify an expression that would contain two identical subexpressions. Doing so also ensures that the value of the common subexpression is computed only once.

```
(+ (* 4 4) (* 4 4)) ⇒ 32

(let ((a (* 4 4))) (+ a a)) ⇒ 32
```

Brackets are often used in place of parentheses to delimit the bindings of a let expression.

```
(let ([(list1 '(a b c)]) [list2 '(d e f)])
  (cons (cons (car list1)
               (car list2))
        (cons (car (cdr list1))
               (car (cdr list2)))))) ⇒ ((a . d) b . e)
```

Scheme treats forms enclosed in brackets just like forms enclosed in parentheses. An open bracket must be matched by a close bracket, and an open parenthesis must be matched by a close parenthesis. We use brackets for let (and, as we'll see, several other standard syntactic forms) to improve readability, especially when we might otherwise have two or more consecutive open parentheses.

Since expressions in the first position of a procedure application are evaluated no differently from other expressions, a let-bound variable may be used there as well.

```
(let ([f +])
  (f 2 3)) ⇒ 5

(let ([f +] [x 2])
  (f x 3)) ⇒ 5

(let ([f +] [x 2] [y 3])
```

Section 2.4. Variables and Let Expressions 23
The variables bound by let are visible only within the body of the let.

\[(f \times y)) \Rightarrow 5\]

\[
\begin{align*}
\text{(let ([+ *])} \\
\quad (+ 2 3)) & \Rightarrow 6 \\
(+ 2 3) & \Rightarrow 5
\end{align*}
\]

This is fortunate, because we would not want the value of + to be the multiplication procedure everywhere.

It is possible to nest let expressions.

\[
\text{(let ([a 4] [b -3])} \\
\quad (\text{let ([a-squared (* a a)])} \\
\quad \quad [b-squared (* b b)]) \\
\quad (+ a-squared b-squared))) \Rightarrow 25
\]

When nested let expressions bind the same variable, only the binding created by the inner let is visible within its body.

\[
\text{(let ([x 1])} \\
\quad (\text{let ([x (+ x 1)])} \\
\quad \quad (+ x x))) \Rightarrow 4
\]

The outer let expression binds x to 1 within its body, which is the second let expression. The inner let expression binds x to (+ x 1) within its body, which is the expression (+ x x). What is the value of (+ x 1)? Since (+ x 1) appears within the body of the outer let but not within the body of the inner let, the value of x must be 1 and hence the value of (+ x 1) is 2. What about (+ x x)? It appears within the body of both let expressions. Only the inner binding for x is visible, so x is 2 and (+ x x) is 4.

The inner binding for x is said to shadow the outer binding. A let-bound variable is visible everywhere within the body of its let expression except where it is shadowed. The region where a variable binding is visible is called its scope. The scope of the first x in the example above is the body of the outer let expression minus the body of the inner let expression, where it is shadowed by the second x. This form of scoping is referred to as lexical scoping, since the scope of each binding can be determined by a straightforward textual analysis of the program.

Shadowing may be avoided by choosing different names for variables. The expression above could be rewritten so that the variable bound by the inner let is new-x.

\[
\text{(let ([x 1])} \\
\quad (\text{let ([new-x (+ x 1)])} \\
\quad \quad (+ new-x new-x))) \Rightarrow 4
\]

Although choosing different names can sometimes prevent confusion, shadowing can help prevent the accidental use of an "old" value. For example, with the original version of the preceding example, it would be impossible for us to mistakenly refer to the outer x within the body of the inner let.

**Exercise 2.4.1**

Rewrite the following expressions, using let to remove common subexpressions and to improve the structure of the code. Do not perform any algebraic simplifications.
a. \((+ (- (* 3 a) b) (+ (* 3 a) b))\)
b. \((\text{cons} (\text{car} (\text{list} a b c)) (\text{cdr} (\text{list} a b c)))\)

Exercise 2.4.2

Determine the value of the following expression. Explain how you derived this value.

\[
\text{(let ([x 9])} \\
\quad (* x \quad \text{(let ([x (/ x 3)])} \quad (+ x x)))
\]

Exercise 2.4.3

Rewrite the following expressions to give unique names to each different \text{let}-bound variable so that none of the variables is shadowed. Verify that the value of your expression is the same as that of the original expression.

a. \((\text{let ([x 'a] [y 'b])} \quad \text{(list} (\text{let ([x 'c]) (cons} x y)) \quad (\text{let ([y 'd]) (cons} x y))))\)
b. \((\text{let ([x '((a b) c)])} \quad \text{(cons} (\text{let ([x (cdr} x)]) \quad \text{(car} x)) \quad \text{(let ([x (car} x)]) \quad \text{(cons} (\text{let ([x (cdr} x)]) \quad \text{(car} x)) \quad \text{(cons} (\text{let ([x (car} x)]) \quad x) \quad (\text{cdr} x)))))))\)

Section 2.5. Lambda Expressions

In the expression \((\text{let ([x (* 3 4)])} \quad (+ x x))\), the variable \(x\) is bound to the value of \((* 3 4)\). What if we would like the value of \((+ x x)\) where \(x\) is bound to the value of \((/ 99 11)\)? Where \(x\) is bound to the value of \((- 2 7)\)? In each case we need a different \text{let} expression. When the body of the \text{let} is complicated, however, having to repeat it can be inconvenient.

Instead, we can use the syntactic form \text{lambda} to create a new procedure that has \(x\) as a parameter and has the same body as the \text{let} expression.

\((\text{lambda} (x) (+ x x)) \Rightarrow #<\text{procedure}>\)

The general form of a \text{lambda} expression is

\((\text{lambda} \quad \text{var ... body}_1 \quad \text{body}_2 \quad ...)\)

The variables \text{var ...} are the \text{formal parameters} of the procedure, and the sequence of expressions \text{body}_1 \quad \text{body}_2 \quad ... is its body. (Actually, the true general form is somewhat more general than this, as you will see later.)

A procedure is just as much an object as a number, string, symbol, or pair. It does not have any meaningful printed representation as far as Scheme is concerned, however, so this book uses the notation
to show that the value of an expression is a procedure.

The most common operation to perform on a procedure is to apply it to one or more values.

\[ \text{((lambda (x) (+ x x)) (* 3 4)) } \Rightarrow 24 \]

This is no different from any other procedure application. The procedure is the value of (lambda (x) (+ x x)), and the only argument is the value of (* 3 4), or 12. The argument values, or actual parameters, are bound to the formal parameters within the body of the lambda expression in the same way as let-bound variables are bound to their values. In this case, \( x \) is bound to 12, and the value of (+ x x) is 24. Thus, the result of applying the procedure to the value 12 is 24.

Because procedures are objects, we can establish a procedure as the value of a variable and use the procedure more than once.

\[
\begin{align*}
\text{(let ((double (lambda (x) (+ x x)))})} \\
\text{(list (double (* 3 4)))} \\
\text{(double (/ 99 11))} \\
\text{(double (- 2 7))),} \Rightarrow (24 \ 18 \ -10)
\end{align*}
\]

Here, we establish a binding for double to a procedure, then use this procedure to double three different values.

The procedure expects its actual parameter to be a number, since it passes the actual parameter on to +. In general, the actual parameter may be any sort of object. Consider, for example, a similar procedure that uses cons instead of +.

\[
\begin{align*}
\text{(let ((double-cons (lambda (x) (cons x x)))})} \\
\text{(double-cons 'a)) } \Rightarrow (a \ . \ a)
\end{align*}
\]

Noting the similarity between double and double-cons, you should not be surprised to learn that they may be collapsed into a single procedure by adding an additional argument.

\[
\begin{align*}
\text{(let ((double-any (lambda (f x) (f x x)))})} \\
\text{(list (double-any + 13))} \\
\text{(double-any cons 'a))),} \Rightarrow (26 \ (a \ . \ a))
\end{align*}
\]

This demonstrates that procedures may accept more than one argument and that arguments passed to a procedure may themselves be procedures.

As with let expressions, lambda expressions become somewhat more interesting when they are nested within other lambda or let expressions.

\[
\begin{align*}
\text{(let ([x 'a])} \\
\text{(let ([f (lambda (y) (list x y))])} \\
\text{(f 'b))) } \Rightarrow (a \ b)
\end{align*}
\]

The occurrence of \( x \) within the lambda expression refers to the \( x \) outside the lambda that is bound by the outer let expression. The variable \( x \) is said to occur free in the lambda expression or to be a free variable of the lambda expression. The variable \( y \) does not occur free in the lambda expression since it is bound by the lambda expression. A variable that occurs free in a lambda expression should be bound, e.g., by an enclosing lambda or let expression, unless the variable is (like the names of primitive procedures) bound outside of the expression, as we discuss in the following section.
What happens when the procedure is applied somewhere outside the scope of the bindings for variables that occur free within the procedure, as in the following expression?

```
(let ([f (let ([x 'sam])
               (lambda (y z) (list x y z)))]
     (f 'i 'am)) ⇒ (sam i am)
```

The answer is that the same bindings that were in effect when the procedure was created are in effect again when the procedure is applied. This is true even if another binding for x is visible where the procedure is applied.

```
(let ([f (let ([x 'sam])
               (lambda (y z) (list x y z)))
     (let ([x 'not-sam])
       (f 'i 'am))⇒ (sam i am)
```

In both cases, the value of x within the procedure named f is sam.

Incidentally, a let expression is nothing more than the direct application of a lambda expression to a set of argument expressions. For example, the two expressions below are equivalent.

```
(let ([x 'a]) (cons x x))
((lambda (x) (cons x x)) 'a)
```

In fact, a let expression is a syntactic extension defined in terms of lambda and procedure application, which are both core syntactic forms. In general, any expression of the form

```
(let ((var expr) ...) body₁ body₂ ...)
```

is equivalent to the following.

```
((lambda (var ...) body₁ body₂ ...) expr ...)
```

See Section 3.1 for more about core forms and syntactic extensions.

As mentioned above, the general form of lambda is a bit more complicated than the form we saw earlier, in that the formal parameter specification, (var ...), need not be a proper list, or indeed even a list at all. The formal parameter specification can be in any of the following three forms:

- a proper list of variables, (var₁ ... varₙ), such as we have already seen,
- a single variable, varᵣ, or
- an improper list of variables, (var₁ ... varₙ . varᵣ).

In the first case, exactly n actual parameters must be supplied, and each variable is bound to the corresponding actual parameter. In the second, any number of actual parameters is valid; all of the actual parameters are put into a single list and the single variable is bound to this list. The third case is a hybrid of the first two cases. At least n actual parameters must be supplied. The variables var₁ ... varₙ are bound to the corresponding actual parameters, and the variable varᵣ is bound to a list containing the remaining actual parameters. In the second and third cases, varᵣ is sometimes referred to as a "rest" parameter because it holds the rest of the actual parameters beyond those that are individually named.

Let's consider a few examples to help clarify the more general syntax of lambda expressions.

```
(let ([f (lambda x x)])
```
(f 1 2 3 4) ⇒ (1 2 3 4)

(let ([f (lambda x x)])
 (f)) ⇒ ()

(let ([g (lambda (x y) (list x y))])
 (g 1 2 3 4)) ⇒ (1 2 3 4)

(let ([h (lambda (x y z) (list x y z))])
 (h 'a 'b 'c 'd)) ⇒ (a b (c d))

In the first two examples, the procedure named $f$ accepts any number of arguments. These arguments are automatically formed into a list to which the variable $x$ is bound; the value of $f$ is this list. In the first example, the arguments are 1, 2, 3, and 4, so the answer is (1 2 3 4). In the second, there are no arguments, so the answer is the empty list () . The value of the procedure named $g$ in the third example is a list whose first element is the first argument and whose second element is a list containing the remaining arguments. The procedure named $h$ is similar but separates out the second argument. While $f$ accepts any number of arguments, $g$ must receive at least one and $h$ must receive at least two.

**Exercise 2.5.1**

Determine the values of the expressions below.

a. (let ([f (lambda (x) x)])
  (f 'a))

b. (let ([f (lambda x x)])
  (f 'a))

c. (let ([f (lambda (x y) x)])
  (f 'a))

d. (let ([f (lambda (x y) y)])
  (f 'a))

**Exercise 2.5.2**

How might the primitive procedure list be defined?

**Exercise 2.5.3**

List the variables that occur free in each of the lambda expressions below. Do not omit variables that name primitive procedures such as + or cons.

a. (lambda (f x) (f x))

b. (lambda (x) (+ x x))

c. (lambda (x y) (f x y))

d. (lambda (x)
  (cons x (f x y)))

e. (lambda (x)
  (let ([z (cons x y)])
   (x y z)))

f. (lambda (x)
  (let ([y (cons x y)])
   (x y z)))
Section 2.6. Top-Level Definitions

The variables bound by let and lambda expressions are not visible outside the bodies of these expressions. Suppose you have created an object, perhaps a procedure, that must be accessible anywhere, like + or cons. What you need is a top-level definition, which may be established with define. Top-level definitions, which are supported by most interactive Scheme systems, are visible in every expression you enter, except where shadowed by another binding.

Let's establish a top-level definition of the double-any procedure of the last section.

```
(define double-any
  (lambda (f x)
    (f x x)))
```

The variable double-any now has the same status as cons or the name of any other primitive procedure. We can use double-any as if it were a primitive procedure.

```
(double-any + 10) ⇒ 20
(double-any cons 'a) ⇒ (a . a)
```

A top-level definition may be established for any object, not just for procedures.

```
(define sandwich "peanut-butter-and-jelly")
```

sandwich ⇒ "peanut-butter-and-jelly"

Most often, though, top-level definitions are used for procedures.

As suggested above, top-level definitions may be shadowed by let or lambda bindings.

```
(define xyz '(x y z))
(let ([xyz '(z y x)])
  xyz) ⇒ (z y x)
```

Variables with top-level definitions act almost as if they were bound by a let expression enclosing all of the expressions you type.

Given only the simple tools you have read about up to this point, it is already possible to define some of the primitive procedures provided by Scheme and described later in this book. If you completed the exercises from the last section, you should already know how to define list.

```
(define list (lambda x x))
```

Also, Scheme provides the abbreviations cadr and cddr for the compositions of car with cdr and cdr with cdr. That is, (cadr list) is equivalent to (car (cdr list)), and, similarly, (cddr list) is equivalent to (cdr (cdr list)). They are easily defined as follows.

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

(define cddr
  (lambda (x)
    (cdr (cdr x))))
```
Any definition (define var expr) where expr is a lambda expression can be written in a shorter form that suppresses the lambda. The exact syntax depends upon the format of the lambda expression's formal parameter specifier, i.e., whether it is a proper list of variables, a single variable, or an improper list of variables. A definition of the form

```
(define var₀
  (lambda (var₁ ... varₙ)
    e₁ e₂ ...))
```

may be abbreviated

```
(define (var₀ var₁ ... varₙ)
  e₁ e₂ ...)
```

while

```
(define var₀
  (lambda varᵣ
    e₁ e₂ ...))
```

may be abbreviated

```
(define (var₀ . varᵣ)
  e₁ e₂ ...)
```

and

```
(define var₀
  (lambda (var₁ ... varₙ . varᵣ)
    e₁ e₂ ...))
```

may be abbreviated

```
(define (var₀ var₁ ... varₙ . varᵣ)
  e₁ e₂ ...)
```

For example, the definitions of cadr and list might be written as follows.

```
(define (cadr x)
  (car (cdr x)))

(define (list . x) x)
```

This book does not often employ this alternative syntax. Although it is shorter, it tends to mask the reality that procedures are not intimately tied to variables, or names, as they are in many other languages. This syntax is often referred to, somewhat pejoratively, as the "defun" syntax for define, after the defun form provided by Lisp languages in which procedures are more closely tied to their names.

Top-level definitions make it easier for us to experiment with a procedure interactively because we need not retype the procedure each time it is used. Let's try defining a somewhat more complicated variation of double-any, one that turns an "ordinary" two-argument procedure into a "doubling" one-argument
procedure.

(define doubler
  (lambda (f)
    (lambda (x) (f x x))))

doubler accepts one argument, \( f \), which must be a procedure that accepts two arguments. The procedure returned by doubler accepts one argument, which it uses for both arguments in an application of \( f \). We can define, with doubler, the simple double and double-cons procedures of the last section.

(define double (doubler +))
(double 13/2) => 13

(define double-cons (doubler cons))
(double-cons 'a) => (a . a)

We can also define double-any with doubler.

(define double-any
  (lambda (f x)
    ((doubler f) x)))

Within double and double-cons, \( f \) has the appropriate value, i.e., + or cons, even though the procedures are clearly applied outside the scope of \( f \).

What happens if you attempt to use a variable that is not bound by a let or lambda expression and that does not have a top-level definition? Try using the variable i-am-not-defined to see what happens.

(i-am-not-defined 3)

Most Scheme systems print a message indicating that an unbound- or undefined-variable exception has occurred.

The system should not, however, complain about the appearance of an undefined variable within a lambda expression, until and unless the resulting procedure is applied. The following should not cause an exception, even though we have not yet established a top-level definition of proc2.

(define procl
  (lambda (x y)
    (proc2 y x)))

If you try to apply procl before defining proc2, you should get a undefined exception message. Let’s give proc2 a top-level definition and try procl.

(define proc2 cons)
(procl 'a 'b) => (b . a)

When you define procl, the system accepts your promise to define proc2, and does not complain unless you use procl before defining proc2. This allows you to define procedures in any order you please. This is especially useful when you are trying to organize a file full of procedure definitions in a way that makes your program more readable. It is necessary when two procedures defined at top level depend upon each other; we will see some examples of this later.
Exercise 2.6.1

What would happen if you were to type

(double-any double-any double-any)

given the definition of double-any from the beginning of this section?

Exercise 2.6.2

A more elegant (though possibly less efficient) way to define `cadr` and `cddr` than given in this section is to define a procedure that composes two procedures to create a third. Write the procedure `compose`, such that `(compose \(p_1\) \(p_2\))` is the composition of \(p_1\) and \(p_2\) (assuming both take one argument). That is, `(compose \(p_1\) \(p_2\))` should return a new procedure of one argument that applies \(p_1\) to the result of applying \(p_2\) to the argument. Use `compose` to define `cadr` and `cddr`.

Exercise 2.6.3

Scheme also provides `caar`, `cdar`, `caaar`, `caadr`, and so on, with any combination of up to four a's (representing `car`) and d's (representing `cdr`) between the c and the r (see Section 6.3). Define each of these with the `compose` procedure of the preceding exercise.

Section 2.7. Conditional Expressions

So far we have considered expressions that perform a given task unconditionally. Suppose that we wish to write the procedure `abs`. If its argument \(x\) is negative, `abs` returns \(-x\); otherwise, it returns \(x\). The most straightforward way to write `abs` is to determine whether the argument is negative and if so negate it, using the `if` syntactic form.

```scheme
(define abs
  (lambda (n)
    (if (< n 0)
        (- 0 n)
        n)))

(abs 77) ⇒ 77
(abs -77) ⇒ 77
```

An `if` expression has the form `(if test consequent alternative)`, where `consequent` is the expression to evaluate if `test` is true and `alternative` is the expression to evaluate if `test` is false. In the expression above, `test` is `< n 0`, `consequent` is `- 0 n`, and `alternative` is `n`.

The procedure `abs` could be written in a variety of other ways. Any of the following are valid definitions of `abs`.

```scheme
(define abs
  (lambda (n)
    (if (>= n 0)
        n
        (- 0 n)))))

(define abs
  (lambda (n)
    (if (not (< n 0))
        n
        (- 0 n))))
```
(define abs  
  (lambda (n)  
    (if (or (> n 0) (= n 0))  
      n  
      (- 0 n)))))

(define abs  
  (lambda (n)  
    (if (= n 0)  
      0  
      (if (< n 0)  
        (- 0 n)  
        n))))

(define abs  
  (lambda (n)  
    ((if (>= n 0) + -)  
      0  
      n))))

The first of these definitions asks if \( n \) is greater than or equal to zero, inverting the test. The second asks if \( n \) is not less than zero, using the procedure not with \(<\). The third asks if \( n \) is greater than zero or \( n \) is equal to zero, using the syntactic form or. The fourth treats zero separately, though there is no benefit in doing so. The fifth is somewhat tricky; \( n \) is either added to or subtracted from zero, depending upon whether \( n \) is greater than or equal to zero.

**Why is if a syntactic form and not a procedure?** In order to answer this, let's revisit the definition of reciprocal from the first section of this chapter.

(define reciprocal  
  (lambda (n)  
    (if (= n 0)  
      "oops!"  
      (/ 1 n)))))

The second argument to the division procedure should not be zero, since the result is mathematically undefined. Our definition of reciprocal avoids this problem by testing for zero before dividing. Were if a procedure, its arguments (including \( (/ 1 n) \)) would be evaluated before it had a chance to choose between the consequent and alternative. Like quote, which does not evaluate its only subexpression, if does not evaluate all of its subexpressions and so cannot be a procedure.

The syntactic form or operates in a manner similar to if. The general form of an or expression is \((or \ expr \ . . .)\). If there are no subexpressions, i.e., the expression is simply \( (or) \), the value is false. Otherwise, each \( expr \) is evaluated in turn until either (a) one of the expressions evaluates to true or (b) no more expressions are left. In case (a), the value is true; in case (b), the value is false.

To be more precise, in case (a), the value of the or expression is the value of the last subexpression evaluated. This clarification is necessary because there are many possible true values. Usually, the value of a test expression is one of the two objects \#t, for true, or \#f, for false.

\(< -1 0 \) \( \Rightarrow \) \#t
Every Scheme object, however, is considered to be either true or false by conditional expressions and by the procedure not. Only #f is considered false; all other objects are considered true.

```
(if #t 'true 'false) ⇒ true
(if #f 'true 'false) ⇒ false
(if '() 'true 'false) ⇒ true
(if '(a b c) 'true 'false) ⇒ true
```

```
(not #t) ⇒ #f
(not "false") ⇒ #f
(not #f) ⇒ #t
```

The and syntactic form is similar in form to or, but an and expression is true if all its subexpressions are true, and false otherwise. In the case where there are no subexpressions, i.e., the expression is simply (and), the value is true. Otherwise, the subexpressions are evaluated in turn until either no more subexpressions are left or the value of a subexpression is false. The value of the and expression is the value of the last subexpression evaluated.

Using and, we can define a slightly different version of reciprocal.

```
(define reciprocal
  (lambda (n)
    (and (not (= n 0))
         (/ 1 n))))
```

```
(reciprocal 3) ⇒ 1/3
(reciprocal 0.5) ⇒ 2.0
(reciprocal 0) ⇒ #f
```

In this version, the value is #f if n is zero and 1/n otherwise.

The procedures =, <, >, <=, and >= are called predicates. A predicate is a procedure that answers a specific question about its arguments and returns one of the two values #t or #f. The names of most predicates end with a question mark (?); the common numeric procedures listed above are exceptions to this rule. Not all predicates require numeric arguments, of course. The predicate null? returns true if its argument is the empty list () and false otherwise.

```
(null? '()) ⇒ #t
(null? 'abc) ⇒ #f
(null? '(x y z)) ⇒ #f
(null? (cdddr '(x y z))) ⇒ #t
```

The procedure cdr must not be passed anything other than a pair, and an exception is raised when this happens. Common Lisp, however, defines (cdr '()) to be (). The following procedure, lisp-cdr, is defined using null? to return () if its argument is ()

---

Section 2.7. Conditional Expressions
(define lisp-cdr
  (lambda (x)
    (if (null? x)
        '()
        (cdr x))))

(lisp-cdr '(a b c)) ⇒ (b c)
(lisp-cdr '(c)) ⇒ ()
(lisp-cdr '()) ⇒ ()

Another useful predicate is eqv?, which requires two arguments. If the two arguments are equivalent, eqv? returns true. Otherwise, eqv? returns false.

(eqv? 'a 'a) ⇒ #t
(eqv? 'a 'b) ⇒ #f
(eqv? #f #f) ⇒ #t
(eqv? #t #t) ⇒ #t
(eqv? #f #t) ⇒ #f
(eqv? 3 3) ⇒ #t
(eqv? 3 2) ⇒ #f
(let ([x "Hi Mom!"])
  (eqv? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
  (eqv? x x)) ⇒ #t
(eqv? (cons 'a 'b) (cons 'a 'b)) ⇒ #f

As you can see, eqv? returns true if the arguments are the same symbol, boolean, number, pair, or string. Two pairs are not the same by eqv? if they are created by different calls to cons, even if they have the same contents. Detailed equivalence rules for eqv? are given in Section 6.2.

Scheme also provides a set of type predicates that return true or false depending on the type of the object, e.g., pair?, symbol?, number?, and string?. The predicate pair?, for example, returns true only if its argument is a pair.

(pair? '(a . c)) ⇒ #t
(pair? '(a b c)) ⇒ #t
(pair? '()) ⇒ #f
(pair? 'abc) ⇒ #f
(pair? "Hi Mom!") ⇒ #f
(pair? 1234567890) ⇒ #f

Type predicates are useful for deciding if the argument passed to a procedure is of the appropriate type. For example, the following version of reciprocal checks first to see that its argument is a number before testing against zero or performing the division.

(define reciprocal
  (lambda (n)
    (if (and (number? n) (not (= n 0)))
        (/ 1 n)
        "oops!")))

(reciprocal 2/3) ⇒ 3/2
(reciprocal 'a) ⇒ "oops!"
By the way, the code that uses reciprocal must check to see that the returned value is a number and not a string. To relieve the caller of this obligation, it is usually preferable to report the error, using assertion-violation, as follows.

```
(define reciprocal
  (lambda (n)
    (if (and (number? n) (not (= n 0)))
      (/ 1 n)
      (assertion-violation 'reciprocal
       "improper argument"
      n))))
```

```
(reciprocal .25) ⇒ 4.0
(reciprocal 0) ⇒ exception in reciprocal: improper argument 0
(reciprocal 'a) ⇒ exception in reciprocal: improper argument a
```

The first argument to assertion-violation is a symbol identifying where the message originates, the second is a string describing the error, and the third and subsequent arguments are "irritants" to be included with the error message.

Let’s look at one more conditional expression, cond, that is often useful in place of if. cond is similar to if except that it allows multiple test and alternative expressions. Consider the following definition of sign, which returns -1 for negative inputs, +1 for positive inputs, and 0 for zero.

```
(define sign
  (lambda (n)
    (if (< n 0)
      -1
      (if (> n 0)
        +1
        0)))))
```

```
(sign -88.3) ⇒ -1
(sign 0) ⇒ 0
(sign 333333333333) ⇒ 1
(* (sign -88.3) (abs -88.3)) ⇒ -88.3
```

The two if expressions may be replaced by a single cond expression as follows.

```
(define sign
  (lambda (n)
    (cond
      [(< n 0) -1]
      [(> n 0) +1]
      [else 0]))))
```

A cond expression usually takes the form

```
(cond (test expr) ... (else expr))
```

though the else clause may be omitted. This should be done only when there is no possibility that all the tests will fail, as in the new version of sign below.

```
(define sign
  (lambda (n)
    (cond
      [(< n 0) -1]
      [(> n 0) +1]
      [else 0]]))
```
These definitions of sign do not depend on the order in which the tests are performed, since only one of the
tests can be true for any value of n. The following procedure computes the tax on a given amount of income
in a progressive tax system with breakpoints at 10,000, 20,000, and 30,000 dollars.

(define income-tax
 (lambda (income)
   (cond
    [((<= income 10000) (* income .05))
     [(<= income 20000) (+ (* (- income 10000) .08) 500.00))
     [(<= income 30000) (+ (* (- income 20000) .13) 1300.00))
     [else (+ (* (- income 30000) .21) 2600.00)])])))

(income-tax 5000) ⇒ 250.0
(income-tax 15000) ⇒ 900.0
(income-tax 25000) ⇒ 1950.0
(income-tax 50000) ⇒ 6800.0

In this example, the order in which the tests are performed, left to right (top to bottom), is significant.

**Exercise 2.7.1**

Define the predicate atom?, which returns true if its argument is not a pair and false if it is.

**Exercise 2.7.2**

The procedure length returns the length of its argument, which must be a list. For example,
(length '(a b c)) is 3. Using length, define the procedure shorter, which returns the shorter of
two list arguments. Have it return the first list if they have the same length.

(shorter '(a b) '(c d e)) ⇒ (a b)
(shorter '(a b) '(c d)) ⇒ (a b)
(shorter '(a b) '(c)) ⇒ (c)

**Section 2.8. Simple Recursion**

We have seen how we can control whether or not expressions are evaluated with if, and, or, and cond. We
can also perform an expression more than once by creating a procedure containing the expression and
invoking the procedure more than once. What if we need to perform some expression repeatedly, say for all
the elements of a list or all the numbers from one to ten? We can do so via recursion. Recursion is a simple
concept: the application of a procedure from within that procedure. It can be tricky to master recursion at first,
but once mastered it provides expressive power far beyond ordinary looping constructs.

A recursive procedure is a procedure that applies itself. Perhaps the simplest recursive procedure is the
following, which we will call goodbye.

(define goodbye
 (lambda ()
   (goodbye)))

Section 2.7. Conditional Expressions
This procedure takes no arguments and simply applies itself immediately. There is no value after the \Rightarrow because goodbye never returns.

Obviously, to make practical use out of a recursive procedure, we must have some way to terminate the recursion. Most recursive procedures should have at least two basic elements, a base case and a recursion step. The base case terminates the recursion, giving the value of the procedure for some base argument. The recursion step gives the value in terms of the value of the procedure applied to a different argument. In order for the recursion to terminate, the different argument must be closer to the base argument in some way.

Let's consider the problem of finding the length of a proper list recursively. We need a base case and a recursion step. The logical base argument for recursion on lists is nearly always the empty list. The length of the empty list is zero, so the base case should give the value zero for the empty list. In order to become closer to the empty list, the natural recursion step involves the cdr of the argument. A nonempty list is one element longer than its cdr, so the recursion step gives the value as one more than the length of the cdr of the list.

\[
\text{(define length}
  \begin{align*}
    &\text{(lambda (ls)}
    \\ &\quad (\text{if (null? ls)}
    \\ &\quad \quad 0
    \\ &\quad \quad (+ \text{ length (cdr ls)) 1}))
  \end{align*}
\]

(length '()) ⇒ 0
(length '(a)) ⇒ 1
(length '(a b)) ⇒ 2

The \textit{if} expression asks if the list is empty. If so, the value is zero. This is the base case. If not, the value is one more than the length of the cdr of the list. This is the recursion step.

Many Scheme implementations allow you to trace the execution of a procedure to see how it operates. In Chez Scheme, for example, one way to trace a procedure is to type \texttt{(trace name)}, where \texttt{name} is the name of a procedure you have defined at top level. If you trace \texttt{length} as defined above and pass it the argument \texttt{'(a b c d)}, you should see something like this:

\[
\begin{align*}
  \text{| (length (a b c d))} \\
  \text{| \quad (length (b c d))} \\
  \text{| \quad \quad (length (c d))} \\
  \text{| \quad \quad \quad (length (d))} \\
  \text{| \quad \quad \quad \quad (length ()))} \\
  \text{| \quad \quad \quad \quad 0} \\
  \text{| \quad \quad \quad \quad 1} \\
  \text{| \quad \quad \quad \quad 2} \\
  \text{| \quad \quad \quad \quad 3} \\
  \text{| \quad \quad \quad \quad 4}
\end{align*}
\]

The indentation shows the nesting level of the recursion; the vertical lines associate applications visually with their values. Notice that on each application of \texttt{length} the list gets smaller until it finally reaches \texttt{()}\:. The value at \texttt{()}\: is 0, and each outer level adds 1 to arrive at the final value.

Let's write a procedure, \texttt{list-copy}, that returns a copy of its argument, which must be a list. That is, \texttt{list-copy} returns a new list consisting of the elements (but not the pairs) of the old list. Making a copy might be useful if either the original list or the copy might be altered via \texttt{set-car!} or \texttt{set-cdr!}, which
we discuss later.

(define list-copy
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (car ls)
              (list-copy (cdr ls)))))

The definition of list-copy is similar to the definition of length. The test in the base case is the same, (null? ls). The value in the base case is (), however, not 0, because we are building up a list, not a number. The recursive call is the same, but instead of adding one, list-copy conses the car of the list onto the value of the recursive call.

There is no reason why there cannot be more than one base case. The procedure memv takes two arguments, an object and a list. It returns the first sublist, or tail, of the list whose car is equal to the object, or #f if the object is not found in the list. The value of memv may be used as a list or as a truth value in a conditional expression.

(define memv
  (lambda (x ls)
    (cond
      [(null? ls) #f]
      [(eqv? (car ls) x) ls]
      [else (memv x (cdr ls))])))

(memv 'a '(a b b d)) ⇒ (a b b d)
(memv 'b '(a b b d)) ⇒ (b b d)
(memv 'c '(a b b d)) ⇒ #f
(memv 'd '(a b b d)) ⇒ (d)
(if (memv 'b '(a b b d))
  "yes"
  "no") ⇒ "yes"

Here there are two conditions to check, hence the use of cond. The first cond clause checks for the base value of (); no object is a member of (), so the answer is #f. The second clause asks if the car of the list is the object, in which case the list is returned, being the first tail whose car contains the object. The recursion step just continues down the list.

There may also be more than one recursion case. Like memv, the procedure remv defined below takes two arguments, an object and a list. It returns a new list with all occurrences of the object removed from the list.

(define remv
  (lambda (x ls)
    (cond
      [(null? ls) '
]
      [(eqv? (car ls) x) (remv x (cdr ls))]
      [else (cons (car ls) (remv x (cdr ls)))])))
This definition is similar to the definition of `memv` above, except `remv` does not quit once it finds the element in the car of the list. Rather, it continues, simply ignoring the element. If the element is not found in the car of the list, `remv` does the same thing as `list-copy` above: it conses the car of the list onto the recursive value.

Up to now, the recursion has been only on the cdr of a list. It is sometimes useful, however, for a procedure to recur on the car as well as the cdr of the list. The procedure `tree-copy` defined below treats the structure of pairs as a tree rather than as a list, with the left subtree being the car of the pair and the right subtree being the cdr of the pair. It performs a similar operation to `list-copy`, building new pairs while leaving the elements (leaves) alone.

```scheme
(define tree-copy
  (lambda (tr)
    (if (not (pair? tr))
        tr
        (cons (tree-copy (car tr))
              (tree-copy (cdr tr))))))
```

The natural base argument for a tree structure is anything that is not a pair, since the recursion traverses pairs rather than lists. The recursive step in this case is doubly recursive, finding the value recursively for the car as well as the cdr of the argument.

At this point, readers who are familiar with other languages that provide special iteration constructs, e.g., `while` or `for` loops, might wonder whether similar constructs are required in Scheme. Such constructs are unnecessary; iteration in Scheme is expressed more clearly and succinctly via recursion. Recursion is more general and eliminates the need for the variable assignments required by many other languages' iteration constructs, resulting in code that is more reliable and easier to follow. Some recursion is essentially iteration and executes as such; Section 3.2 has more to say about this. Often, there is no need to make a distinction, however. Concentrate instead on writing clear, concise, and correct programs.

Before we leave the topic of recursion, let's consider a special form of repetition called mapping. Consider the following procedure, `abs-all`, that takes a list of numbers as input and returns a list of their absolute values.

```scheme
(define abs-all
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (abs (car ls))
              (abs-all (cdr ls))))))
```

This procedure forms a new list from the input list by applying the procedure `abs` to each element. We say that `abs-all` maps `abs` over the input list to produce the output list. Mapping a procedure over a list is a fairly common thing to do, so Scheme provides the procedure `map`, which maps its first argument, a procedure, over its second, a list. We can use `map` to define `abs-all`.

```scheme
(define abs-all
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (abs (car ls))
              (abs-all (cdr ls))))))
```

This procedure forms a new list from the input list by applying the procedure `abs` to each element. We say that `abs-all` maps `abs` over the input list to produce the output list. Mapping a procedure over a list is a fairly common thing to do, so Scheme provides the procedure `map`, which maps its first argument, a procedure, over its second, a list. We can use `map` to define `abs-all`.

```scheme
(define abs-all
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (abs (car ls))
              (abs-all (cdr ls))))))
```

This procedure forms a new list from the input list by applying the procedure `abs` to each element. We say that `abs-all` maps `abs` over the input list to produce the output list. Mapping a procedure over a list is a fairly common thing to do, so Scheme provides the procedure `map`, which maps its first argument, a procedure, over its second, a list. We can use `map` to define `abs-all`.
(lambda (ls)
  (map abs ls)))

We really do not need abs-all, however, since the corresponding direct application of map is just as short and perhaps clearer.

(map abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)

Of course, we can use lambda to create the procedure argument to map, e.g., to square the elements of a list of numbers.

(map (lambda (x) (* x x))
     '(1 -3 -5 7)) ⇒ (1 9 25 49)

We can map a multiple-argument procedure over multiple lists, as in the following example.

(map cons '(a b c) '(1 2 3)) ⇒ ((a . 1) (b . 2) (c . 3))

The lists must be of the same length, and the procedure should accept as many arguments as there are lists. Each element of the output list is the result of applying the procedure to corresponding members of the input list.

Looking at the first definition of abs-all above, you should be able to derive, before studying it, the following definition of map1, a restricted version of map that maps a one-argument procedure over a single list.

(define map1
  (lambda (p ls)
    (if (null? ls)
        '()
        (cons (p (car ls))
              (map1 p (cdr ls))))))

(map1 abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)

All we have done is to replace the call to abs in abs-all with a call to the new parameter p. A definition of the more general map is given in Section 5.4.

Exercise 2.8.1

Describe what would happen if you switched the order of the arguments to cons in the definition of tree-copy.

Exercise 2.8.2

Consult Section 6.3 for the description of append and define a two-argument version of it. What would happen if you switched the order of the arguments in the call to append within your definition of append?

Exercise 2.8.3

Define the procedure make-list, which takes a nonnegative integer n and an object and returns a new list, n long, each element of which is the object.

(make-list 7 '()) ⇒ ((() () () () () () ()())

Section 2.8. Simple Recursion
[Hint: The base test should be \((n \leq 0)\), and the recursion step should involve \((- n - 1)\). Whereas \((\_\) is the natural base case for recursion on lists, 0 is the natural base case for recursion on nonnegative integers. Similarly, subtracting 1 is the natural way to bring a nonnegative integer closer to 0.]

**Exercise 2.8.4**

The procedures `list-ref` and `list-tail` return the \(n\)th element and \(n\)th tail of a list \(ls\).

\[
\begin{align*}
\text{(list-ref '(1 2 3 4) 0)} & \Rightarrow 1 \\
\text{(list-tail '(1 2 3 4) 0)} & \Rightarrow (1 2 3 4) \\
\text{(list-ref '(a short (nested) list) 2)} & \Rightarrow \text{(nested)} \\
\text{(list-tail '(a short (nested) list) 2)} & \Rightarrow \text{((nested) list)}
\end{align*}
\]

Define both procedures.

**Exercise 2.8.5**

Exercise 2.7.2 had you use `length` in the definition of `shorter`, which returns the shorter of its two list arguments, or the first if the two have the same length. Write `shorter` without using `length`. [Hint: Define a recursive helper, `shorter?`, and use it in place of the length comparison.]

**Exercise 2.8.6**

All of the recursive procedures shown so far have been directly recursive. That is, each procedure directly applies itself to a new argument. It is also possible to write two procedures that use each other, resulting in indirect recursion. Define the procedures `odd?` and `even?`, each in terms of the other. [Hint: What should each return when its argument is 0?]

\[
\begin{align*}
\text{(even? 17)} & \Rightarrow \text{#f} \\
\text{(odd? 17)} & \Rightarrow \text{#t}
\end{align*}
\]

**Exercise 2.8.7**

Use `map` to define a procedure, `transpose`, that takes a list of pairs and returns a pair of lists as follows.

\[
\text{(transpose '(((a . 1) (b . 2) (c . 3)))) } \Rightarrow (\text{(a b c) 1 2 3})
\]

[Hint: \((\text{(a b c) 1 2 3})\) is the same as \((\text{(a b c) . (1 2 3)})\).]

**Section 2.9. Assignment**

Although many programs can be written without them, assignments to top-level variables or `let`-bound and `lambda`-bound variables are sometimes useful. Assignments do not create new bindings, as with `let` or `lambda`, but rather change the values of existing bindings. Assignments are performed with `set!`.

\[
\begin{align*}
\text{(define abcde '(a b c d e))} \\
abcde & \Rightarrow (\text{a b c d e}) \\
\text{(set! abcde (cdr abcde))} \\
abcde & \Rightarrow (\text{b c d e}) \\
\text{(let ([abcde '(a b c d e)])} \\
\text{ (set! abcde (reverse abcde))} \\
abcde & \Rightarrow (\text{e d c b a})
\end{align*}
\]
Many languages require the use of assignments to initialize local variables, separate from the declaration or binding of the variables. In Scheme, all local variables are given a value immediately upon binding. Besides making the separate assignment to initialize local variables unnecessary, it ensures that the programmer cannot forget to initialize them, a common source of errors in most languages.

In fact, most of the assignments that are either necessary or convenient in other languages are both unnecessary and inconvenient in Scheme, since there is typically a clearer way to express the same algorithm without assignments. One common practice in some languages is to sequence expression evaluation with a series of assignments, as in the following procedure that finds the roots of a quadratic equation.

```
(define quadratic-formula
  (lambda (a b c)
    (let ([root1 0] [root2 0] [minusb 0] [radical 0] [divisor 0])
      (set! minusb (- 0 b))
      (set! radical (sqrt (- (* b b) (* 4 (* a c)))))
      (set! divisor (* 2 a))
      (set! root1 (/ (+ minusb radical) divisor))
      (set! root2 (/ (- minusb radical) divisor))
    (cons root1 root2))))
```

The roots are computed according to the well-known quadratic formula,

\[x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\]

which yields the solutions to the equation \(0 = ax^2 + bx + c\). The \texttt{let} expression in this definition is employed solely to establish the variable bindings, corresponding to the declarations required in other languages. The first three assignment expressions compute subpieces of the formula, namely \(-b\), \(\sqrt{b^2 - 4ac}\), and \(2a\). The last two assignment expressions compute the two roots in terms of the subpieces. A pair of the two roots is the value of \texttt{quadratic-formula}. For example, the two roots of \(2x^2 - 4x - 6\) are \(x = 3\) and \(x = -1\).

```
(quadratic-formula 2 -4 -6) \Rightarrow (3 . -1)
```

The definition above works, but it can be written more clearly without the assignments, as shown below.

```
(define quadratic-formula
  (lambda (a b c)
    (let ([minusb (- 0 b)]
          [radical (sqrt (- (* b b) (* 4 (* a c))))]
          [divisor (* 2 a)])
      (let ([root1 (/ (+ minusb radical) divisor)]
            [root2 (/ (- minusb radical) divisor)])
        (cons root1 root2))))
```

In this version, the \texttt{set!} expressions are gone, and we are left with essentially the same algorithm. By employing two \texttt{let} expressions, however, the definition makes clear the dependency of \texttt{root1} and \texttt{root2} on the values of \texttt{minusb}, \texttt{radical}, and \texttt{divisor}. Equally important, the \texttt{let} expressions make clear the lack of dependencies among \texttt{minusb}, \texttt{radical}, and \texttt{divisor} and between \texttt{root1} and \texttt{root2}.

Assignments do have some uses in Scheme, otherwise the language would not support them. Consider the following version of \texttt{cons} that counts the number of times it is called, storing the count in a variable named \texttt{cons-count}. It uses \texttt{set!} to increment the count; there is no way to achieve the same behavior without assignments.

```
(define kons-count 0)
```
(define kons
    (lambda (x y)
        (set! kons-count (+ kons-count 1))
        (cons x y)))

(kons 'a '(b c))  \Rightarrow  (a b c)
kons-count  \Rightarrow  1

(kons 'a (kons 'b (kons 'c '())))  \Rightarrow  (a b c)
kons-count  \Rightarrow  4

Assignments are commonly used to implement procedures that must maintain some internal state. For example, suppose we would like to define a procedure that returns 0 the first time it is called, 1 the second time, 2 the third time, and so on indefinitely. We could write something similar to the definition of cons-count above:

(define next 0)
(define count
    (lambda ()
        (let ([v next])
            (set! next (+ next 1))
            v)))

(count)  \Rightarrow  0
(count)  \Rightarrow  1

This solution is somewhat undesirable in that the variable next is visible at top level even though it need not be. Since it is visible at top level, any code in the system can change its value, perhaps inadvertently affecting the behavior of count in a subtle way. We can solve this problem by let-binding next outside of the lambda expression:

(define count
    (let ([next 0])
        (lambda ()
            (let ([v next])
                (set! next (+ next 1))
                v))))

The latter solution also generalizes easily to provide multiple counters, each with its own local counter. The procedure make-counter, defined below, returns a new counting procedure each time it is called.

(define make-counter
    (lambda ()
        (let ([next 0])
            (lambda ()
                (let ([v next])
                    (set! next (+ next 1))
                    v)))))

Since next is bound inside of make-counter but outside of the procedure returned by make-counter, each procedure it returns maintains its own unique counter.

(define count1 (make-counter))
(define count2 (make-counter))
If a state variable must be shared by more than one procedure defined at top level, but we do not want the state variable to be visible at top level, we can use \texttt{let} to bind the variable and \texttt{set!} to make the procedures visible at top level.

\begin{verbatim}
(define shhh #f)
(define tell #f)
(let ([secret 0])
  (set! shhh
    (lambda (message)
      (set! secret message)))
  (set! tell
    (lambda ()
      secret)))

(shhh "sally likes harry")
(tell) ⇒ "sally likes harry"
(secret ⇒ exception: variable secret is not bound)
\end{verbatim}

Variables must be defined before they can be assigned, so we define \texttt{shhh} and \texttt{tell} to be \texttt{#f} initially. (Any initial value would do.) We'll see this structure again in Section 3.5 and a better way to structure code like this as a library in Section 3.6.

Local state is sometimes useful for caching computed values or allowing a computation to be evaluated \textit{lazily}, i.e., only once and only on demand. The procedure \texttt{lazy} below accepts a \texttt{thunk}, or zero-argument procedure, as an argument. Thunks are often used to "freeze" computations that must be delayed for some reason, which is exactly what we need to do in this situation. When passed a thunk \texttt{t}, \texttt{lazy} returns a new thunk that, when invoked, returns the value of invoking \texttt{t}. Once computed, the value is saved in a local variable so that the computation need not be performed again. A boolean flag is used to record whether \texttt{t} has been invoked and its value saved.

\begin{verbatim}
(define lazy
  (lambda (t)
    (let ([val #f] [flag #f])
      (lambda ()
        (if (not flag)
            (begin (set! val (t))
                   (set! flag #t))
            val)))))
\end{verbatim}

The syntactic form \texttt{begin}, used here for the first time, evaluates its subexpressions in sequence from left to right and returns the value of the last subexpression, like the body of a \texttt{let} or \texttt{lambda} expression. We also see that the \textit{alternative} subexpression of an \texttt{if} expression can be omitted. This should be done only when the value of the \texttt{if} is discarded, as it is in this case.

Lazy evaluation is especially useful for values that require considerable time to compute. By delaying the evaluation, we might avoid computing the value altogether, and by saving the value, we avoid computing it more than once.
The operation of `lazy` can best be illustrated by printing a message from within a thunk passed to `lazy`.

```
(define p
  (lazy (lambda ()
    (display "Ouch!"
    (newline)
    "got me"))

The first time `p` is invoked, the message `Ouch!` is printed and the string "got me" is returned. Thereafter, "got me" is returned but the message is not printed. The procedures `display` and `newline` are the first examples of explicit input/output we have seen; `display` prints the string without quotation marks, and `newline` prints a newline character.

To further illustrate the use of `set!`, let’s consider the implementation of stack objects whose internal workings are not visible on the outside. A stack object accepts one of four messages: `empty?`, which returns `#t` if the stack is empty; `push!`, which adds an object to the top of the stack; `top`, which returns the object on the top of the stack; and `pop!`, which removes the object on top of the stack. The procedure `make-stack` given below creates a new stack each time it is called in a manner similar to `make-counter`.

```
(define make-stack
  (lambda ()
    (let ((ls '()))
      (lambda (msg . args)
        (cond
          [(eqv? msg 'empty?) (null? ls)]
          [(eqv? msg 'push!) (set! ls (cons (car args) ls))]
          [(eqv? msg 'top) (car ls)]
          [(eqv? msg 'pop!) (set! ls (cdr ls))]
          [else "oops"]))))

Each stack is stored as a list bound to the variable `ls`; `set!` is used to change this binding for `push!` and `pop!`. Notice that the argument list of the inner `lambda` expression uses the improper list syntax to bind `args` to a list of all arguments but the first. This is useful here because in the case of `empty?`, `top`, and `pop!` there is only one argument (the message), but in the case of `push!` there are two (the message and the object to push onto the stack).

```
(define stack1 (make-stack))
(define stack2 (make-stack))
(list (stack1 'empty?) (stack2 'empty?)) ⇒ (#t #t)

(stack1 'push! 'a)
(list (stack1 'empty?) (stack2 'empty?)) ⇒ (#f #t)

(stack1 'push! 'b)
(stack2 'push! 'c)
(stack1 'top) ⇒ b
(stack2 'top) ⇒ c

(stack1 'pop!)
(stack1 'top) ⇒ a
(list (stack1 'empty?) (stack2 'empty?)) ⇒ (#f #f)

(stack1 'pop!)
```
As with the counters created by `make-counter`, the state maintained by each stack object is directly accessible only within the object. Each reference or change to this state is made explicitly by the object itself. One important benefit is that we can change the internal structure of the stack, perhaps to use a vector (see Section 6.9) instead of a list to hold the elements, without changing its external behavior. Because the behavior of the object is known abstractly (not operationally), it is known as an *abstract object*. See Section 12.8 for more about creating abstract objects.

In addition to changing the values of variables, we can also change the values of the car and cdr fields of a pair, using the procedures `set-car!` and `set-cdr!`.

\[
\text{(define p (list 1 2 3))}
\]
\[
\text{(set-car! (cdr p) 'two)}
\]
\[
p \Rightarrow (1 \text{ two } 3)
\]
\[
\text{(set-cdr! p '())}
\]
\[
p \Rightarrow (1)
\]

We can use these operators to define a queue data type, which is like a stack except that new elements are added at one end and extracted from the other. The following queue implementation uses a `tconc` structure. A tconc consists of a nonempty list and a header. The header is a pair whose car points to the first pair (head) of the list and whose cdr points to the last pair (end) of the list.

The last element of the list is a placeholder and not considered part of the queue.

Four operations on queues are defined below: `make-queue`, which constructs a queue; `putq!`, which adds an element to the end of a queue; `getq`, which retrieves the element at the front of a queue; and `delq!`, which removes the element at the front of a queue.

\[
\text{(define make-queue}
\]
\[
\text{ (lambda ()}
\]
\[
\text{ (let ([end (cons 'ignored '())])}
\]
\[
\text{ (cons end end)))}
\]
\[
\text{(define putq!}
\]
\[
\text{ (lambda (q v)}
\]
\[
\text{ (let ([end (cons 'ignored '())])}
\]
\[
\text{ (set-car! (cdr q) v)}
\]
\[
\text{ (set-cdr! (cdr q) end)}
\]
\[
\text{ (set-cdr! q end)))}
\]
\[
\text{(define getq}
\]
\[
\text{ (lambda (q)}
\]
\[
\text{ (car (car q)))}
\]
\[
\text{(define delq!}
\]
\[
\text{ (lambda (q)}
\]
\[
\text{ (set-car! q (cdr (car q))))}
\]
All are simple operations except for \texttt{putq!}, which modifies the end pair to contain the new value and adds a new end pair.

\begin{verbatim}
(define myq (make-queue))

(putq! myq 'a)
(putq! myq 'b)
(getq myq) \Rightarrow a
(delq! myq)
(getq myq) \Rightarrow b
(delq! myq)
(putq! myq 'c)
(putq! myq 'd)
(getq myq) \Rightarrow c
(delq! myq)
(getq myq) \Rightarrow d
\end{verbatim}

\textbf{Exercise 2.9.1}

Modify \texttt{make-counter} to take two arguments: an initial value for the counter to use in place of 0 and an amount to increment the counter by each time.

\textbf{Exercise 2.9.2}

Look up the description of \texttt{case} in Section 5.3. Replace the \texttt{cond} expression in \texttt{make-stack} with an equivalent \texttt{case} expression. Add \texttt{mt?} as a second name for the \texttt{empty?} message.

\textbf{Exercise 2.9.3}

Modify the \texttt{stack} object to allow the two messages \texttt{ref} and \texttt{set!}. \texttt{(stack 'ref i)} should return the \texttt{i}th element from the top of the stack; \texttt{(stack 'ref 0)} should be equivalent to \texttt{(stack 'top)}. \texttt{(stack 'set! i v)} should change the \texttt{i}th element from the top of the stack to \texttt{v}.

\begin{verbatim}
(define stack (make-stack))

(stack 'push! 'a)
(stack 'push! 'b)
(stack 'push! 'c)

(stack 'ref 0) \Rightarrow c
(stack 'ref 2) \Rightarrow a
(stack 'set! 1 'd)
(stack 'ref 1) \Rightarrow d
(stack 'top) \Rightarrow c
(stack 'pop!)
(stack 'top) \Rightarrow d
\end{verbatim}

[Hint: Use \texttt{list-ref} to implement \texttt{ref} and \texttt{list-tail} with \texttt{set-car!} to implement \texttt{set!}.

\textbf{Exercise 2.9.4}

Scheme supports \texttt{vectors} as well as lists. Like lists, vectors are aggregate objects that contain other objects. Unlike lists, vectors have a fixed size and are laid out in one flat block of memory, typically with a header containing the length of the vector, as in the ten-element vector below.

```
```

48

Section 2.9. Assignment
This makes vectors more suitable for applications needing fast access to any element of the aggregate but less suitable for applications needing data structures that grow and shrink as needed.

Look up the basic vector operations in Section 6.9 and reimplement the stack object to use a vector instead of a list to hold the stack contents. Include the ref and set! messages of Exercise 2.9.3. Have the new make-stack accept a size argument \( n \) and make the vector length \( n \), but do not otherwise change the external (abstract) interface.

**Exercise 2.9.5**

Define a predicate, emptyq?, for determining if a queue is empty. Modify getq and delq! to raise an exception when an empty queue is found, using assertion-violation.

**Exercise 2.9.6**

In the queue implementation, the last pair in the encapsulated list is a placeholder, i.e., it never holds anything useful. Recode the queue operators to avoid this wasted pair. Make sure that the series of queue operations given earlier works with the new implementation. Which implementation do you prefer?

**Exercise 2.9.7**

Using set-cdr!, it is possible to create cyclic lists. For example, the following expression evaluates to a list whose car is the symbol \( a \) and whose cdr is the list itself.

\[
\text{(let } ([ls (cons 'a '())]) \\
\text{ (set-cdr! ls ls)} \\
\text{ ls)}
\]

What happens when you enter the above expression during an interactive Scheme session? What will the implementation of length on page 42 do when given a cyclic list? What does the built-in length primitive do?

**Exercise 2.9.8**

Define the predicate list?, which returns \( \#t \) if its argument is a proper list and \( \#f \) otherwise (see Section 6.3). It should return \( \#f \) for cyclic lists as well as for lists terminated by objects other than \( () \).

\[
\text{(list? '()) } \Rightarrow \#t \\
\text{(list? '(1 2 3)) } \Rightarrow \#t \\
\text{(list? '(a . b)) } \Rightarrow \#f \\
\text{(list? (let ([ls (cons 'a '())]) \\
\text{ (set-cdr! ls ls)}) \\
\text{ ls)) } \Rightarrow \#f
\]

First write a simplified version of list? that does not handle cyclic lists, then extend this to handle cyclic lists correctly. Revise your definition until you are satisfied that it is as clear and concise as possible. [Hint: Use the following "hare and tortoise" algorithm to detect cycles. Define a recursive help procedure of two arguments, the hare and the tortoise. Start both the hare and the tortoise at the beginning of the list. Have the hare advance by two cdrs each time the tortoise advances by one cdr. If the hare catches the tortoise, there must be a cycle.]
Chapter 3. Going Further

The preceding chapter prepared you to write Scheme programs using a small set of the most useful primitive syntactic forms and procedures. This chapter introduces a number of additional features and programming techniques that will allow you to write more sophisticated and efficient programs.

Section 3.1. Syntactic Extension

As we saw in Section 2.5, the let syntactic form is merely a syntactic extension defined in terms of a lambda expression and a procedure application, both core syntactic forms. At this point, you might be wondering which syntactic forms are core forms and which are syntactic extensions, and how new syntactic extensions may be defined. This section provides some answers to these questions.

In truth, it is not necessary for us to draw a distinction between core forms and syntactic extensions, since once defined, a syntactic extension has exactly the same status as a core form. Drawing a distinction, however, makes understanding the language easier, since it allows us to focus attention on the core forms and to understand all others in terms of them.

It is necessary for a Scheme implementation to distinguish between core forms and syntactic extensions. A Scheme implementation expands syntactic extensions into core forms as the first step of compilation or interpretation, allowing the rest of the compiler or interpreter to focus only on the core forms. The set of core forms remaining after expansion to be handled directly by the compiler or interpreter is implementation-dependent, however, and may be different from the set of forms described as core here.

The exact set of syntactic forms making up the core of the language is thus subject to debate, although it must be possible to derive all other forms from any set of forms declared to be core forms. The set described here is among the simplest for which this constraint is satisfied.

The core syntactic forms include top-level define forms, constants, variables, procedure applications, quote expressions, lambda expressions, if expressions, and set! expressions. The grammar below describes the core syntax of Scheme in terms of these definitions and expressions. In the grammar, vertical bars (|) separate alternatives, and a form followed by an asterisk (*) represents zero or more occurrences of the form. <variable> is any Scheme identifier. <datum> is any Scheme object, such as a number, list, symbol, or vector. <boolean> is either #t or #f, <number> is any number, <character> is any character, and <string> is any string. We have already seen examples of numbers, strings, lists, symbols, and booleans. See Chapter 6 or the formal syntax description starting on page 455 for more on the object-level syntax of these and other objects.

```
<program>  → <form>*
<form>    → <definition> | <expression>
<definition> → (define <variable> <expression>)
<expression> → <constant>
Γ <variable>
Γ (quote <datum>)
Γ (lambda <formals> <expression> <expression>*)
Γ (if <expression> <expression> <expression>)
Γ (set! <variable> <expression>)
Γ <application>
<constant> → <boolean> | <number> | <character> | <string>
<formals> → <variable> 
```
The grammar is ambiguous in that the syntax for procedure applications conflicts with the syntaxes for quote, lambda, if, and set! expressions. In order to qualify as a procedure application, the first <expression> must not be one of these keywords, unless the keyword has been redefined or locally bound.

The "defun" syntax for define given in Section 2.6 is not included in the core, since definitions in that form are straightforwardly translated into the simpler define syntax. Similarly, the core syntax for if does not permit the alternative to be omitted, as did one example in Section 2.9. An if expression lacking an alternative can be translated into the core syntax for if merely by replacing the missing subexpression with an arbitrary constant, such as #f.

A begin that contains only definitions is considered to be a definition in the grammar; this is permitted in order to allow syntactic extensions to expand into more than one definition. begin expressions, i.e., begin forms containing expressions, are not considered core forms. A begin expression of the form

\( (\text{begin} \ e_1 \ e_2 \ ...) \)

is equivalent to the lambda application

\( ((\lambda () \ e_1 \ e_2 \ ...)) \)

and hence need not be considered core.

Now that we have established a set of core syntactic forms, let's turn to a discussion of syntactic extensions. Syntactic extensions are so called because they extend the syntax of Scheme beyond the core syntax. All syntactic extensions in a Scheme program must ultimately be derived from the core forms. One syntactic extension, however, may be defined in terms of another syntactic extension, as long as the latter is in some sense "closer" to the core syntax. Syntactic forms may appear anywhere an expression or definition is expected, as long as the extended form expands into a definition or expression as appropriate.

Syntactic extensions are defined with define-syntax. define-syntax is similar to define, except that define-syntax associates a syntactic transformation procedure, or transformer, with a keyword (such as let), rather than associating a value with a variable. Here is how we might define let with define-syntax.

\[
\begin{align*}
\text{(define-syntax let} \\
\quad \text{(syntax-rules ()} \\
\quad \quad \left[ \_ ((x e) ...) b1 b2 ...} \\
\quad \quad \quad \quad ((\lambda (x ...) b1 b2 ...) e ...)]]
\end{align*}
\]

The identifier appearing after define-syntax is the name, or keyword, of the syntactic extension being defined, in this case let. The syntax-rules form is an expression that evaluates to a transformer. The item following syntax-rules is a list of auxiliary keywords and is nearly always ( ). An example of an auxiliary keyword is the else of cond. (Other examples requiring the use of auxiliary keywords are given in Chapter 8.) Following the list of auxiliary keywords is a sequence of one or more rules, or pattern/template pairs. Only one rule appears in our definition of let. The pattern part of a rule specifies the form that the input must take, and the template specifies to what the input should be transformed.

The pattern should always be a structured expression whose first element is an underscore (_). (As we will see in Chapter 8, the use of _ is only a convention, but it is a good one to follow.) If more than one rule is present, the appropriate one is chosen by matching the patterns, in order, against the input during expansion. It is a syntax violation if none of the patterns match the input.
Identifiers other than an underscore or ellipsis appearing within a pattern are pattern variables, unless they are listed as auxiliary keywords. Pattern variables match any substructure and are bound to that substructure within the corresponding template. The notation \texttt{pat} \ldots in the pattern allows for zero or more expressions matching the ellipsis prototype \texttt{pat} in the input. Similarly, the notation \texttt{expr} \ldots in the template produces zero or more expressions from the ellipsis prototype \texttt{expr} in the output. The number of \texttt{pats} in the input determines the number of \texttt{exprs} in the output; in order for this to work, any ellipsis prototype in the template must contain at least one pattern variable from an ellipsis prototype in the pattern.

The single rule in our definition of \texttt{let} should be fairly self-explanatory, but a few points are worth mentioning. First, the syntax of \texttt{let} requires that the body contain at least one form; hence, we have specified \texttt{b1 b2 \ldots} instead of \texttt{b \ldots}, which might seem more natural. On the other hand, \texttt{let} does not require that there be at least one variable/value pair, so we were able to use, simply, \texttt{(x e) \ldots}. Second, the pattern variables \texttt{x} and \texttt{e}, though together within the same prototype in the pattern, are separated in the template; any sort of rearrangement or recombination is possible. Finally, the three pattern variables \texttt{x}, \texttt{e}, and \texttt{b2} that appear in ellipsis prototypes in the pattern also appear in ellipsis prototypes in the template. This is not a coincidence; it is a requirement. In general, if a pattern variable appears within an ellipsis prototype in the pattern, it cannot appear outside an ellipsis prototype in the template.

The definition of \texttt{and} below is somewhat more complex than the one for \texttt{let}.

\begin{verbatim}
(define-syntax and
  (syntax-rules ()
    [(_ _) #t]
    [(_ e) e]
    [(_ e1 e2 e3 ...)
      (if e1 (and e2 e3 ...) #f)]))
\end{verbatim}

This definition is recursive and involves more than one rule. Recall that \texttt{(and)} evaluates to \texttt{#t}; the first rule takes care of this case. The second and third rules specify the base case and recursion steps of the recursion and together translate \texttt{and} expressions with two or more subexpressions into nested \texttt{if} expressions. For example, \texttt{(and a b c)} expands first into

\begin{verbatim}
(if a (and b c) #f)
\end{verbatim}

then

\begin{verbatim}
(if a (if b (and c) #f) #f)
\end{verbatim}

and finally

\begin{verbatim}
(if a (if b c #f) #f)
\end{verbatim}

With this expansion, if \texttt{a} and \texttt{b} evaluate to a true value, then the value is the value of \texttt{c}, otherwise \texttt{#f}, as desired.

The version of \texttt{and} below is simpler but, unfortunately, incorrect.

\begin{verbatim}
(define-syntax and ; incorrect!
  (syntax-rules ()
    [(_ _) #t]
    [(_ e1 e2 ...)
      (if e1 (and e2 e3 ...) #f)]))
\end{verbatim}

The expression

\begin{verbatim}
(if a (and b c) #f)
\end{verbatim}
\( \text{(and \ (not \ (= \ x \ 0)) \ (/ \ 1 \ x))} \)

should return the value of \( (/ \ 1 \ x) \) when \( x \) is not zero. With the incorrect version of \text{and}, the expression expands as follows.

\[
\begin{align*}
(\text{if} \ (\text{not} \ (= \ x \ 0)) \ (\text{and} \ (/ \ 1 \ x)) \ #f) & \rightarrow \\
(\text{if} \ (\text{not} \ (= \ x \ 0)) \ (\text{if} \ (/ \ 1 \ x) \ (\text{and}) \ #f) \ #f) & \rightarrow \\
(\text{if} \ (\text{not} \ (= \ x \ 0)) \ (\text{if} \ (/ \ 1 \ x) \ #t \ #f) \ #f)
\end{align*}
\]

The final answer if \( x \) is not zero is \#t, not the value of \( (/ \ 1 \ x) \).

The definition of \text{or} below is similar to the one for \text{and} except that a temporary variable must be introduced for each intermediate value so that we can both test the value and return it if it is a true value. (A temporary variable is not needed for \text{and} since there is only one false value, \#f.)

\[
\text{(define-syntx or} \\
\text{(syntax-rules ()} \\
\text{[(_) #f]} \\
\text{[(_ e) e]} \\
\text{[(_ e1 e2 e3 ...)]} \\
\text{(let ([t e1])} \\
\text{\quad (if t t (or e2 e3 ...))))})
\]

Like variables bound by \text{lambda} or \text{let}, identifiers introduced by a template are lexically scoped, i.e., visible only within expressions introduced by the template. Thus, even if one of the expressions \( e2 \ e3 \ldots \) contains a reference to \( t \), the introduced binding for \( t \) does not "capture" those references. This is typically accomplished via automatic renaming of introduced identifiers.

As with the simpler version of \text{and} given above, the simpler version of \text{or} below is incorrect.

\[
\text{(define-syntx or ; incorrect!} \\
\text{(syntax-rules ()} \\
\text{[(_) #f]} \\
\text{[(_ e1 e2 ...) \\
\text{\quad (let ([t e1])} \\
\text{\quad \quad (if t t (or e2 e3 ...))))})}
\]

The reason is more subtle, however, and is the subject of Exercise 3.2.6.

**Exercise 3.1.1**

Write out the expansion steps necessary to expand

\[
\text{(let ([x (memv 'a ls)]) (and x (memv 'b x))})
\]

into core forms.

**Exercise 3.1.2**

Write out the expansion steps necessary to expand

\[
\text{(or (memv x '(a b c)) (list x))}
\]
Exercise 3.1.3

let* is similar to let but evaluates its bindings in sequence. Each of the right-hand-side expressions is within the scope of the earlier bindings.

\[
\text{let* } \{[a\ 5] \[b\ (+\ a\ a)] \[c\ (+\ a\ b)]\} \\
\quad \text{(list } a\ b\ c) \Rightarrow (5\ 10\ 15)
\]

let* can be implemented as nested let expressions. For example, the let* expression above is equivalent to the nested let expressions below.

\[
\text{let } \{[a\ 5]\} \\
\quad \text{let } \{[b\ (+\ a\ a)]\} \\
\quad \text{let } \{[c\ (+\ a\ b)]\} \\
\quad \text{(list } a\ b\ c)\}} \Rightarrow (5\ 10\ 15)
\]

Define let* with define-syntax.

Exercise 3.1.4

As we saw in Section 2.9, it is legal to omit the third, or alternative, subexpression of an if expression. Doing so, however, often leads to confusion. Scheme provides two syntactic forms, when and unless, that may be used in place of such "one-armed" if expressions.

\[
\text{when } \text{test } \text{expr}_1 \text{ expr}_2 \ldots \\
\text{unless } \text{test } \text{expr}_1 \text{ expr}_2 \ldots 
\]

With both forms, test is evaluated first. For when, if test evaluates to true, the remaining forms are evaluated in sequence as if enclosed in an implicit begin expression. If test evaluates to false, the remaining forms are not evaluated, and the result is unspecified. unless is similar except that the remaining forms are evaluated only if test evaluates to false.

\[
\text{let } \{[x\ 3]\} \\
\quad \text{unless } (=\ x\ 0) \ (\text{set! } x\ (+\ x\ 1)) \\
\quad \text{(when } (=\ x\ 4) \ (\text{set! } x\ (*\ x\ 2))) \\
\quad x \Rightarrow 8
\]

Define when as a syntactic extension in terms of if and begin, and define unless in terms of when.

Section 3.2. More Recursion

In Section 2.8, we saw how to define recursive procedures using top-level definitions. Before that, we saw how to create local bindings for procedures using let. It is natural to wonder whether a let-bound procedure can be recursive. The answer is no, at least not in a straightforward way. If you try to evaluate the expression

\[
\text{let } \{[\text{sum} \ (\lambda (ls) \\
\quad \text{(if } (\text{null? } ls) \\
\quad \quad 0 \\
\quad \quad (+ \ (\text{car } ls) \ (\text{sum } (\text{cdr } ls))))))\} \\
\quad (\text{sum } '(1\ 2\ 3\ 4\ 5)))
\]
it will probably raise an exception with a message to the effect that \texttt{sum} is undefined. This is because the variable \texttt{sum} is visible only within the body of the \texttt{let} expression and not within the \texttt{lambda} expression whose value is bound to \texttt{sum}. We can get around this problem by passing the procedure \texttt{sum} to itself as follows.

\begin{verbatim}
(let ([sum (lambda (sum ls)
            (if (null? ls)
                0
                (+ (car ls) (sum sum (cdr ls)))))])
  (sum sum '(1 2 3 4 5))) ⇒ 15
\end{verbatim}

This works and is a clever solution, but there is an easier way, using \texttt{letrec}. Like \texttt{let}, the \texttt{letrec} syntactic form includes a set of variable-value pairs, along with a sequence of expressions referred to as the \textit{body} of the \texttt{letrec}.

\begin{verbatim}
(letrec (((var expr) ...) body₁ body₂ ...)

Unlike \texttt{let}, the variables \texttt{var} ... are visible not only within the body of the \texttt{letrec} but also within \texttt{expr} .... Thus, we can rewrite the expression above as follows.

\begin{verbatim}
(letrec ([sum (lambda (ls)
            (if (null? ls)
                0
                (+ (car ls) (sum (cdr ls)))))])
  (sum '(1 2 3 4 5))) ⇒ 15
\end{verbatim}

Using \texttt{letrec}, we can also define mutually recursive procedures, such as the procedures \texttt{even?} and \texttt{odd?} that were the subject of Exercise 2.8.6.

\begin{verbatim}
(letrec ([even? (lambda (x)
                (or (= x 0)
                    (odd? (- x 1)))]
        [odd? (lambda (x)
                (and (not (= x 0))
                    (even? (- x 1)))]
  (list (even? 20) (odd? 20))) ⇒ (#t #f)
\end{verbatim}

In a \texttt{letrec} expression, \texttt{expr} ... are most often \texttt{lambda} expressions, though this need not be the case. One restriction on the expressions must be obeyed, however. It must be possible to evaluate each \texttt{expr} without evaluating any of the variables \texttt{var} .... This restriction is always satisfied if the expressions are all \texttt{lambda} expressions, since even though the variables may appear within the \texttt{lambda} expressions, they cannot be evaluated until the resulting procedures are invoked in the body of the \texttt{letrec}. The following \texttt{letrec} expression obeys this restriction.

\begin{verbatim}
(letrec ([f (lambda () (+ x 2))]
    [x 1])
  (f)) ⇒ 3
\end{verbatim}

while the following does not.

\begin{verbatim}
(letrec ([y (+ x 2)]
    [x 1])
\end{verbatim}
In this case, an exception is raised indicating that \( x \) is not defined where it is referenced.

We can use \texttt{letrec} to hide the definitions of "help" procedures so that they do not clutter the top-level namespace. This is demonstrated by the definition of \texttt{list?} below, which follows the "hare and tortoise" algorithm outlined in Exercise 2.9.8.

\[
\text{(define list?}
\text{  (lambda (x)
    (letrec ([race
      (lambda (h t)
        (if (pair? h)
          (let ([h (cdr h)])
            (if (pair? h)
              (and (not (eq? h t))
                (race (cdr h) (cdr t)))
              (null? h)))
          (null? h))))
    (race x x))))}
\]

When a recursive procedure is called in only one place outside the procedure, as in the example above, it is often clearer to use a \textit{named} \texttt{let} expression. Named \texttt{let} expressions take the following form.

\[
\text{(let name ((var expr) ...) body_1 body_2 ...)}
\]

Named \texttt{let} is similar to unnamed \texttt{let} in that it binds the variables \texttt{var} \ldots to the values of \texttt{expr} \ldots within the body \texttt{body_1 body_2} \ldots. As with unnamed \texttt{let}, the variables are visible only within the body and not within \texttt{expr} \ldots. In addition, the variable \texttt{name} is bound within the body to a procedure that may be called to recur; the arguments to the procedure become the new values for the variables \texttt{var} \ldots.

The definition of \texttt{list?} has been rewritten below to use named \texttt{let}.

\[
\text{(define list?}
\text{  (lambda (x)
    (let race ([h x] [t x])
      (if (pair? h)
        (let ([h (cdr h)])
          (if (pair? h)
            (and (not (eq? h t))
              (race (cdr h) (cdr t)))
            (null? h)))
        (null? h))))
\]

Just as \texttt{let} can be expressed as a simple direct application of a \texttt{lambda} expression to arguments, named \texttt{let} can be expressed as the application of a recursive procedure to arguments. A named \texttt{let} of the form

\[
\text{(let name ((var expr) ...) body_1 body_2 ...)}
\]

can be rewritten in terms of \texttt{letrec} as follows.

\[
\text{((letrec ((name (lambda (var ...)) body_1 body_2 ...)))}
\]
Alternatively, it can be rewritten as

\[
\text{(letrec ((name (lambda (var ... ) body_1 body_2 ...)))} \\
\text{(name expr ...))}
\]

provided that the variable name does not appear free within expr ....

As we discussed in Section 2.8, some recursion is essentially iteration and executes as such. When a procedure call is in tail position (see below) with respect to a lambda expression, it is considered to be a tail call, and Scheme systems must treat it properly, as a "goto" or jump. When a procedure tail-calls itself or calls itself indirectly through a series of tail calls, the result is tail recursion. Because tail calls are treated as jumps, tail recursion can be used for indefinite iteration in place of the more restrictive iteration constructs provided by other programming languages, without fear of overflowing any sort of recursion stack.

A call is in tail position with respect to a lambda expression if its value is returned directly from the lambda expression, i.e., if nothing is left to do after the call but to return from the lambda expression. For example, a call is in tail position if it is the last expression in the body of a lambda expression, the consequent or alternative part of an if expression in tail position, the last subexpression of an and or or expression in tail position, the last expression in the body of a let or letrec in tail position, etc. Each of the calls to \text{f} in the expressions below are tail calls, but the calls to \text{g} are not.

\[
\begin{align*}
\text{(lambda () (f (g)))} \\
\text{(lambda () (if (g) (f) (f)))} \\
\text{(lambda () (let ([x 4]) (f)))} \\
\text{(lambda () (or (g) (f)))}
\end{align*}
\]

In each case, the values of the calls to \text{f} are returned directly, whereas the calls to \text{g} are not.

Recursion in general and named \text{let} in particular provide a natural way to implement many algorithms, whether iterative, recursive, or partly iterative and partly recursive; the programmer is not burdened with two distinct mechanisms.

The following two definitions of factorial use named \text{let} expressions to compute the factorial, \( n! \), of a nonnegative integer \( n \). The first employs the recursive definition \( n! = n \times (n - 1)! \), where \( 0! \) is defined to be 1.

\[
\begin{align*}
\text{(define factorial} \\
\text{(lambda (n) } \\
\text{ (let fact ([i n])} \\
\text{ (if (= i 0) } \\
\text{ 1} \\
\text{ (* i (fact (- i 1)))))))}
\end{align*}
\]

\[
\begin{align*}
\text{(factorial 0)} & \Rightarrow 1 \\
\text{(factorial 1)} & \Rightarrow 1 \\
\text{(factorial 2)} & \Rightarrow 2 \\
\text{(factorial 3)} & \Rightarrow 6 \\
\text{(factorial 10)} & \Rightarrow 3628800
\end{align*}
\]

The second is an iterative version that employs the iterative definition \( n! = n \times (n - 1) \times (n - 2) \times ... \times 1 \), using an accumulator, \( a \), to hold the intermediate products.
A similar problem is to compute the $n$th Fibonacci number for a given $n$. The Fibonacci numbers are an infinite sequence of integers, 0, 1, 1, 2, 3, 5, 8, etc., in which each number is the sum of the two preceding numbers in the sequence. A procedure to compute the $n$th Fibonacci number is most naturally defined recursively as follows.

```
(define fibonacci
  (lambda (n)
    (if (= n 0) 0
        (let fib ([i n])
          (cond
            [(= i 0) 0]
            [(= i 1) 1]
            [else (+ (fib (- i 1)) (fib (- i 2)))])))))
```

This solution requires the computation of the two preceding Fibonacci numbers at each step and hence is **doubly recursive**. For example, to compute $\texttt{(fibonacci 4)}$ requires the computation of both $\texttt{(fib 3)}$ and $\texttt{(fib 2)}$, to compute $\texttt{(fib 3)}$ requires computing both $\texttt{(fib 2)}$ and $\texttt{(fib 1)}$, and to compute $\texttt{(fib 2)}$ requires computing both $\texttt{(fib 1)}$ and $\texttt{(fib 0)}$. This is very inefficient, and it becomes more inefficient as $n$ grows. A more efficient solution is to adapt the accumulator solution of the factorial example above to use two accumulators, $a1$ for the current Fibonacci number and $a2$ for the preceding one.

```
(define fibonacci
  (lambda (n)
    (if (= n 0) 0
        (let fib ([i n] [a1 1] [a2 0])
          (if (= i 1) a1
              (fib (- i 1) (+ a1 a2) a1)))))
```

Here, zero is treated as a special case, since there is no preceding value. This allows us to use the single base case $\texttt{(fib 0)}$. The time it takes to compute the $n$th Fibonacci number using this iterative solution grows linearly with $n$, which makes a significant difference when compared to the doubly recursive version. To get a feel for the difference, try computing $\texttt{(fibonacci 35)}$ and $\texttt{(fibonacci 40)}$ using both definitions to see how long each takes.

We can also get a feel for the difference by looking at a trace for each on small inputs. The first trace below shows the calls to $\texttt{fib}$ in the non-tail-recursive version of $\texttt{fibonacci}$, with input 5.
Notice how there are several calls to \( \text{fib} \) with arguments 2, 1, and 0. The second trace shows the calls to \( \text{fib} \) in the tail-recursive version, again with input 5.

\[
\begin{array}{l}
(fib 5) \\
(fib 4) \\
|(fib 3) \\
| (fib 2) \\
| | (fib 1) \\
| | 1 \\
| | (fib 0) \\
| | 0 \\
| 1 \\
| (fib 1) \\
| 1 \\
| 2 \\
| (fib 2) \\
| (fib 1) \\
| 1 \\
| (fib 0) \\
| 0 \\
| 1 \\
| (fib 1) \\
| 1 \\
| 2 \\
| 5 \\
\end{array}
\]

Clearly, there is quite a difference.

The named \textit{let} examples shown so far are either tail-recursive or not tail-recursive. It often happens that one recursive call within the same expression is tail-recursive while another is not. The definition of \textit{factor} below computes the prime factors of its nonnegative integer argument. The first call to \( f \) is not tail-recursive, but the second one is.

\begin{verbatim}
(define factor
  (lambda (n)
    (let f ([n n] [i 2])
      (cond
        [ (> i n) (list n)]
        [(integer? (/ n i))
          (cons (factor i) (f n (* i i)))]
        [else (f n (+ i 1))]))))
\end{verbatim}
\begin{equation}
\text{(cons \(i\) (f \(\ell n i\) \(i\)))}
\text{[else (f \(n\) (+ \(i\) 1)))])})
\end{equation}

\begin{align*}
\text{(factor 0)} & \Rightarrow (0) \\
\text{(factor 1)} & \Rightarrow (1) \\
\text{(factor 12)} & \Rightarrow (2 \ 2 \ 3) \\
\text{(factor 3628800)} & \Rightarrow (2 \ 2 \ 2 \ 2 \ 2 \ 2 \ 2 \ 3 \ 3 \ 3 \ 3 \ 5 \ 5 \ 7) \\
\text{(factor 9239)} & \Rightarrow (9239)
\end{align*}

A trace of the calls to \(f\), produced in Chez Scheme by replacing \textit{let} with \textit{trace-let}, in the evaluation of (factor 120) below highlights the difference between the nontail calls and the tail calls.

\begin{verbatim}
(f 120 2)
| (f 60 2)
| | (f 30 2)
| | | (f 15 2)
| | | | (f 15 3)
| | | | | (f 5 3)
| | | | | | (f 5 4)
| | | | | | | (f 5 5)
| | | | | | | | | (5)
| | | | | | | | | | (3 5)
| | | | | | | | | | | (2 3 5)
| | | | | | | | | | | | (2 2 3 5)
| | | | | | | | | | | | | (2 2 2 3 5)
\end{verbatim}

A nontail call to \(f\) is shown indented relative to its caller, since the caller is still active, whereas tail calls appear at the same level of indentation.

**Exercise 3.2.1**

Which of the recursive procedures defined in Section 3.2 are tail-recursive, and which are not?

**Exercise 3.2.2**

Rewrite \textit{factor} using \textit{letrec} to bind \(f\) in place of named \textit{let}. Which version do you prefer?

**Exercise 3.2.3**

Can the \textit{letrec} expression below be rewritten using named \textit{let}? If not, why not? If so, do it.

\begin{verbatim}
(letrec ([(even? \\
          (lambda (x) \\
            (or (= x 0) \\
               (odd? (- x 1))))])
         [(odd? \\
          (lambda (x) \\
            (and (not (= x 0)) \\
                 (even? (- x 1))))])
         (even? 20))
\end{verbatim}
Exercise 3.2.4

Rewrite both definitions of fibonacci given in this section to count the number of recursive calls to fib, using a counter similar to the one used in the cons-count example of Section 2.9. Count the number of recursive calls made in each case for several input values. What do you notice?

Exercise 3.2.5

Augment the definition of let given in Section 3.1 to handle named let as well as unnamed let, using two rules.

Exercise 3.2.6

The following definition of or is simpler than the one given in Section 3.1.

```
(define-syntax or ; incorrect!
  (syntax-rules ()
    [(_ #f)]
    [(_ e1 e2 ...) (let ([t e1])
                    (if t t (or e2 ...)))])
)
```

Say why it is not correct. [Hint: Think about what would happen if this version of or were used in the even? and odd? example given on page 66 for very large inputs.]

Exercise 3.2.7

The definition of factor is not the most efficient possible. First, no factors of n besides n itself can possibly be found beyond \(\sqrt{n}\). Second, the division \((/ n i)\) is performed twice when a factor is found. Third, after 2, no even factors can possibly be found. Recode factor to correct all three problems. Which is the most important problem to solve? Are there any additional improvements you can make?

Section 3.3. Continuations

During the evaluation of a Scheme expression, the implementation must keep track of two things: (1) what to evaluate and (2) what to do with the value. Consider the evaluation of \((\text{null? x})\) within the expression below.

```
(if (null? x) (quote ()) (cdr x))
```

The implementation must first evaluate \((\text{null? x})\) and, based on its value, evaluate either \((\text{quote ()})\) or \((\text{cdr x})\). "What to evaluate" is \((\text{null? x})\), and "what to do with the value" is to make the decision which of \((\text{quote ()})\) and \((\text{cdr x})\) to evaluate and to do so. We call "what to do with the value" the continuation of a computation.

Thus, at any point during the evaluation of any expression, there is a continuation ready to complete, or at least continue, the computation from that point. Let's assume that \(x\) has the value \((a b c)\). We can isolate six continuations during the evaluation of \((\text{if (null? x) (quote ()) (cdr x)})\), the continuations waiting for

1. the value of \((\text{if (null? x) (quote ()) (cdr x)})\),
2. the value of \((\text{null? x})\),
3. the value of \(\text{null?}\),
4. the value of \(x\),
5. the value of `cdr`, and
6. the value of `x` (again).

The continuation of `(cdr x)` is not listed because it is the same as the one waiting for
`(if (null? x) (quote ()) (cdr x)).

Scheme allows the continuation of any expression to be captured with the procedure `call/cc`. `call/cc` must be passed a procedure `p` of one argument. `call/cc` constructs a concrete representation of the current continuation and passes it to `p`. The continuation itself is represented by a procedure `k`. Each time `k` is applied to a value, it returns the value to the continuation of the `call/cc` application. This value becomes, in essence, the value of the application of `call/cc`.

If `p` returns without invoking `k`, the value returned by the procedure becomes the value of the application of `call/cc`.

Consider the simple examples below.

```scheme
(call/cc
 (lambda (k)
    (* 5 4))))  \⇒  20
```

```scheme
(call/cc
 (lambda (k)
    (* 5 (k 4))))  \⇒  4
```

```scheme
(+ 2
 (call/cc
 (lambda (k)
    (* 5 (k 4))))))  \⇒  6
```

In the first example, the continuation is captured and bound to `k`, but `k` is never used, so the value is simply the product of 5 and 4. In the second, the continuation is invoked before the multiplication, so the value is the value passed to the continuation, 4. In the third, the continuation includes the addition by 2; thus, the value is the value passed to the continuation, 4, plus 2.

Here is a less trivial example, showing the use of `call/cc` to provide a nonlocal exit from a recursion.

```scheme
(define product
 (lambda (ls)
    (call/cc
     (lambda (break)
      (let f ([ls ls])
        (cond
          [(null? ls) 1]
          [(= (car ls) 0) (break 0)]
          [else (* (car ls) (f (cdr ls))))])))
     )))

(product '(1 2 3 4 5))  \⇒  120
(product '(7 3 8 0 1 9 5))  \⇒  0
```

The nonlocal exit allows `product` to return immediately, without performing the pending multiplications, when a zero value is detected.

Each of the continuation invocations above returns to the continuation while control remains within the
procedure passed to call/cc. The following example uses the continuation after this procedure has already returned.

(let ((x (call/cc (lambda (k) k))))
  (x (lambda (ignore) "hi")))  \Rightarrow "hi"

The continuation captured by this invocation of call/cc may be described as "Take the value, bind it to x, and apply the value of x to the value of (lambda (ignore) "hi")." Since (lambda (k) k) returns its argument, x is bound to the continuation itself; this continuation is applied to the procedure resulting from the evaluation of (lambda (ignore) "hi"). This has the effect of binding x (again!) to this procedure and applying the procedure to itself. The procedure ignores its argument and returns "hi".

The following variation of the example above is probably the most confusing Scheme program of its size; it might be easy to guess what it returns, but it takes some thought to figure out why.

((((call/cc (lambda (k) k)) (lambda (x) x)) "HEY!") \Rightarrow "HEY!"

The value of the call/cc is its own continuation, as in the preceding example. This is applied to the identity procedure (lambda (x) x), so the call/cc returns a second time with this value. Then, the identity procedure is applied to itself, yielding the identity procedure. This is finally applied to "HEY!", yielding "HEY!".

Continuations used in this manner are not always so puzzling. Consider the following definition of factorial that saves the continuation at the base of the recursion before returning 1, by assigning the top-level variable retry.

(define retry #f)

(define factorial
  (lambda (x)
    (if (= x 0)
      (call/cc (lambda (k) (set! retry k) 1))
      (* x (factorial (- x 1))))))

With this definition, factorial works as we expect factorial to work, except it has the side effect of assigning retry.

(factorial 4) \Rightarrow 24
(retry 1) \Rightarrow 24
(retry 2) \Rightarrow 48

The continuation bound to retry might be described as "Multiply the value by 1, then multiply this result by 2, then multiply this result by 3, then multiply this result by 4." If we pass the continuation a different value, i.e., not 1, we will cause the base value to be something other than 1 and hence change the end result.

(retry 2) \Rightarrow 48
(retry 5) \Rightarrow 120

This mechanism could be the basis for a breakpoint package implemented with call/cc; each time a breakpoint is encountered, the continuation of the breakpoint is saved so that the computation may be restarted from the breakpoint (more than once, if desired).

Continuations may be used to implement various forms of multitasking. The simple "light-weight process" mechanism defined below allows multiple computations to be interleaved. Since it is nonpreemptive, it
requires that each process voluntarily "pause" from time to time in order to allow the others to run.

(define lwp-list '())
(define lwp
  (lambda (thunk)
    (set! lwp-list (append lwp-list (list thunk)))))

(define start
  (lambda ()
    (let ([p (car lwp-list)])
      (set! lwp-list (cdr lwp-list))
      (p))))

(define pause
  (lambda ()
    (call/cc
      (lambda (k)
        (lwp (lambda () (k #f)))
        (start)))))

The following light-weight processes cooperate to print an infinite sequence of lines containing "hey!".

(lwp (lambda () (let f () (pause) (display "h") (f))))
(lwp (lambda () (let f () (pause) (display "e") (f))))
(lwp (lambda () (let f () (pause) (display "y") (f))))
(lwp (lambda () (let f () (pause) (display "!") (f))))
(lwp (lambda () (let f () (pause) (newline) (f))))

(start) ⇒ hey!
  hey!
  hey!
  hey!
  →

See Section 12.11 for an implementation of engines, which support preemptive multitasking, with call/cc.

Exercise 3.3.1

Use call/cc to write a program that loops indefinitely, printing a sequence of numbers beginning at zero. Do not use any recursive procedures, and do not use any assignments.

Exercise 3.3.2

Rewrite product without call/cc, retaining the feature that no multiplications are performed if any of the list elements are zero.

Exercise 3.3.3

What would happen if a process created by lwp as defined above were to terminate, i.e., simply return without calling pause? Define a quit procedure that allows a process to terminate without otherwise affecting the lwp system. Be sure to handle the case in which the only remaining process terminates.
Exercise 3.3.4

Each time \texttt{lwp} is called, the list of processes is copied because \texttt{lwp} uses \texttt{append} to add its argument to the end of the process list. Modify the original \texttt{lwp} code to use the queue data type developed in Section 2.9 to avoid this problem.

Exercise 3.3.5

The light-weight process mechanism allows new processes to be created dynamically, although the example given in this section does not do so. Design an application that requires new processes to be created dynamically and implement it using the light-weight process mechanism.

Section 3.4. Continuation Passing Style

As we discussed in the preceding section, a continuation waits for the value of each expression. In particular, a continuation is associated with each procedure call. When one procedure invokes another via a nontail call, the called procedure receives an implicit continuation that is responsible for completing what is left of the calling procedure's body plus returning to the calling procedure's continuation. If the call is a tail call, the called procedure simply receives the continuation of the calling procedure.

We can make the continuations explicit by encapsulating "what to do" in an explicit procedural argument passed along on each call. For example, the continuation of the call to \texttt{f} in

\begin{verbatim}
(letrec ([f (lambda (x) (cons 'a x))]
    [g (lambda (x) (cons 'b (f x)))]
    [h (lambda (x) (g (cons 'c x)))]
    (cons 'd (h '())))) ⇒ (d b a c)
\end{verbatim}

conses the symbol \texttt{b} onto the value returned to it, then returns the result of this cons to the continuation of the call to \texttt{g}. This continuation is the same as the continuation of the call to \texttt{h}, which conses the symbol \texttt{d} onto the value returned to it. We can rewrite this in \textit{continuation-passing style}, or CPS, by replacing these implicit continuations with explicit procedures.

\begin{verbatim}
(letrec ([f (lambda (x k) (k (cons 'a x)))]
    [g (lambda (x k)
        (f x (lambda (v) (k (cons 'b v)))))]
    [h (lambda (x k) (g (cons 'c x) k))])
    (h '() (lambda (v) (cons 'd v))))
\end{verbatim}

Like the implicit continuation of \texttt{h} and \texttt{g} in the preceding example, the explicit continuation passed to \texttt{h} and on to \texttt{g},

\begin{verbatim}
(lambda (v) (cons 'd v))
\end{verbatim}

conses the symbol \texttt{d} onto the value passed to it. Similarly, the continuation passed to \texttt{f},

\begin{verbatim}
(lambda (v) (k (cons 'b v)))
\end{verbatim}

conses \texttt{b} onto the value passed to it, then passes this on to the continuation of \texttt{g}.

Expressions written in CPS are more complicated, of course, but this style of programming has some useful applications. CPS allows a procedure to pass more than one result to its continuation, because the procedure that implements the continuation can take any number of arguments.
(define car&cdr
  (lambda (p k)
    (k (car p) (cdr p))))

(car&cdr '(a b c)
  (lambda (x y)
    ((b c) a))
(car&cdr '(a b c) cons)
(car&cdr '(a b c a d) memv)

(This can be done with multiple values as well; see Section 5.8.) CPS also allows a procedure to take separate "success" and "failure" continuations, which may accept different numbers of arguments. An example is integer-divide below, which passes the quotient and remainder of its first two arguments to its third, unless the second argument (the divisor) is zero, in which case it passes an error message to its fourth argument.

(define integer-divide
  (lambda (x y success failure)
    (if (= y 0)
      (failure "divide by zero")
      (let ((q (quotient x y)))
        (success q (- x (* q y)))))))

(integer-divide 10 3 list (lambda (x) x)) ⇒ (3 1)
(integer-divide 10 0 list (lambda (x) x)) ⇒ "divide by zero"

The procedure quotient, employed by integer-divide, returns the quotient of its two arguments, truncated toward zero.

Explicit success and failure continuations can sometimes help to avoid the extra communication necessary to separate successful execution of a procedure from unsuccessful execution. Furthermore, it is possible to have multiple success or failure continuations for different flavors of success or failure, each possibly taking different numbers and types of arguments. See Sections 12.10 and 12.11 for extended examples that employ continuation-passing style.

At this point you might be wondering about the relationship between CPS and the continuations captured via call/cc. It turns out that any program that uses call/cc can be rewritten in CPS without call/cc, but a total rewrite of the program (sometimes including even system-defined primitives) might be necessary. Try to convert the product example on page 75 into CPS before looking at the version below.

(define product
  (lambda (ls k) (lambda (x y) (k (car ls) (cdr ls)))))

(product '(1 2 3 4 5) (lambda (x) x)) ⇒ 120
(product '(7 3 8 0 1 9 5) (lambda (x) x)) ⇒ 0
Exercise 3.4.1

Rewrite the reciprocal example first given in Section 2.1 to accept both success and failure continuations, like integer-divide above.

Exercise 3.4.2

Rewrite the retry example from page 75 to use CPS.

Exercise 3.4.3

Rewrite the following expression in CPS to avoid using call/cc.

\[
\text{(define reciprocals}
\begin{array}{l}
\quad \text{(lambda (ls)} \\
\quad \quad \text{(call/cc}
\begin{array}{l}
\quad \quad \quad \text{(lambda (k)} \\
\quad \quad \quad \quad \text{(map (lambda (x)} \\
\quad \quad \quad \quad \quad \text{(if (= x 0)} \\
\quad \quad \quad \quad \quad \quad \text{(k "zero found")} \\
\quad \quad \quad \quad \quad \quad (/* 1 x))}
\quad \quad \quad \quad \quad \quad \text{ls})))))
\end{array}
\end{array}
\right)
\]

\[
\text{(reciprocals '(2 1/3 5 1/4)) \Rightarrow (1/2 3 1/5 4)}
\]
\[
\text{(reciprocals '(2 1/3 0 5 1/4)) \Rightarrow "zero found"}
\]

[Hint: A single-list version of map is defined on page 46.]

Section 3.5. Internal Definitions

In Section 2.6, we discussed top-level definitions. Definitions may also appear at the front of a lambda, let, or letrec body, in which case the bindings they create are local to the body.

\[
\text{(define f (lambda (x) (* x x))}
\begin{array}{l}
\quad \text{(let ([x 3])}
\quad \quad \text{(define f (lambda (y) (+ y x))}
\quad \quad \quad \text{(f 4)) \Rightarrow 7}
\quad \quad \quad \text{(f 4) \Rightarrow 16}
\end{array}
\right)
\]

Procedures bound by internal definitions can be mutually recursive, as with letrec. For example, we can rewrite the even? and odd? example from Section 3.2 using internal definitions as follows.

\[
\text{(let ()}
\begin{array}{l}
\quad \text{(define even?}
\begin{array}{l}
\quad \quad \text{(lambda (x)} \\
\quad \quad \quad \text{(or (= x 0)} \\
\quad \quad \quad \quad \text{(odd? (- x 1))))})
\end{array}
\right)
\]
\[
\text{(define odd?}
\begin{array}{l}
\quad \quad \text{(lambda (x)} \\
\quad \quad \quad \text{(and (not (= x 0)} \\
\quad \quad \quad \quad \text{(even? (- x 1))))})
\end{array}
\right)
\]

\[
\text{(even? 20)) \Rightarrow \#t}
\]
Similarly, we can replace the use of \texttt{letrec} to bind \texttt{race} with an internal definition of \texttt{race} in our first definition of \texttt{list}?

\begin{verbatim}
(define list?
  (lambda (x)
    (define race
      (lambda (h t)
        (if (pair? h)
            (let ([h (cdr h)])
              (if (pair? h)
                  (and (not (eq? h t))
                    (race (cdr h) (cdr t)))
                  (null? h)))
            (null? h)))
    (race x x)))
\end{verbatim}

In fact, internal variable definitions and \texttt{letrec} are practically interchangeable. The only difference, other than the obvious difference in syntax, is that variable definitions are guaranteed to be evaluated from left to right, while the bindings of a letrec may be evaluated in any order. So we cannot quite replace a \texttt{lambda}, \texttt{let}, or \texttt{letrec} body containing internal definitions with a \texttt{letrec} expression. We can, however, use \texttt{letrec*}, which, like \texttt{let*}, guarantees left-to-right evaluation order. A body of the form

\begin{verbatim}
(letrec* ((var expr\textsubscript{0}) ... expr\textsubscript{1} expr\textsubscript{2} ...)
\end{verbatim}

is equivalent to a \texttt{letrec*} expression binding the defined variables to the associated values in a body comprising the expressions.

\begin{verbatim}
(letrec* ((var expr\textsubscript{0}) ... expr\textsubscript{1} expr\textsubscript{2} ...)
\end{verbatim}

Conversely, a \texttt{letrec*} of the form

\begin{verbatim}
(letrec* ((var expr\textsubscript{0}) ... expr\textsubscript{1} expr\textsubscript{2} ...)
\end{verbatim}

can be replaced with a \texttt{let} expression containing internal definitions and the expressions from the body as follows.

\begin{verbatim}
(let ()
  (define var expr\textsubscript{0})
  ... expr\textsubscript{1}
  expr\textsubscript{2}
  ...)
\end{verbatim}

The seeming lack of symmetry between these transformations is due to the fact that \texttt{letrec*} expressions can appear anywhere an expression is valid, whereas internal definitions can appear only at the front of a body. Thus, in replacing a \texttt{letrec*} with internal definitions, we must generally introduce a \texttt{let} expression to hold the definitions.

Another difference between internal definitions and \texttt{letrec} or \texttt{letrec*} is that syntax definitions may

\textbf{Section 3.5. Internal Definitions}
appear among the internal definitions, while \texttt{letrec} and \texttt{letrec*} bind only variables.

\begin{verbatim}
(let ([x 3])
  (define-syntax set-x!
    (syntax-rules ()
      [(_ e) (set! x e)])
  (set-x! (+ x x))
)x) \Rightarrow 6
\end{verbatim}

The scope of a syntactic extension established by an internal syntax definition, as with an internal variable
definition, is limited to the body in which the syntax definition appears.

Internal definitions may be used in conjunction with top-level definitions and assignments to help modularize
programs. Each module of a program should make visible only those bindings that are needed by other
modules, while hiding other bindings that would otherwise clutter the top-level namespace and possibly result
in unintended use or redefinition of those bindings. A common way of structuring a module is shown below.

\begin{verbatim}
(define export-var #f)
;
(let ()
  (define var expr)
  ;
  init-expr
  ;
  (set! export-var export-val)
  ;
)
\end{verbatim}

The first set of definitions establish top-level bindings for the variables we desire to export (make visible
globally). The second set of definitions establish local bindings visible only within the module. The
expressions \textit{init-expr} ... perform any initialization that must occur after the local bindings have been
established. Finally, the \texttt{set!} expressions assign the exported variables to the appropriate values.

An advantage of this form of modularization is that the bracketing \texttt{let} expression may be removed or
"commented out" during program development, making the internal definitions top-level to facilitate
interactive testing. This form of modularization also has several disadvantages, as we discuss in the next
section.

The following module exports a single variable, \texttt{calc}, which is bound to a procedure that implements a
simple four-function calculator.

\begin{verbatim}
(define calc #f)
(let ()
  (define do-calc
    (lambda (ek expr)
      (cond
        ;[(number? expr) expr]
        ;[(and (list? expr) (= (length expr) 3))
             (let ([op (car expr)] [args (cdr expr)])
               (case op
                 ;[(add) (apply-op ek + args)]
                 ;[(sub) (apply-op ek - args)]
                 ;[(mul) (apply-op ek * args)]
                 ;[(div) (apply-op ek / args)]
\end{verbatim}
(define apply-op
  (lambda (ek op args)
    (op (do-calc ek (car args)) (do-calc ek (cadr args))))
)

(define complain
  (lambda (ek msg expr)
    (ek (list msg expr)))))

(set! calc
  (lambda (expr)
    ; grab an error continuation ek
    (call/cc
      (lambda (ek)
        (do-calc ek expr))))))

(calc '(add (mul 3 2) -4)) ⇒ 2
(calc '(div 1/2 1/6)) ⇒ 3
(calc '(add (mul 3 2) (div 4))) ⇒ ("invalid expression" (div 4))
(calc '(mul (add 1 -2) (pow 2 7))) ⇒ ("invalid operator" pow)

This example uses a case expression to determine which operator to apply. case is similar to cond except that the test is always the same: (memv val (key ...)), where val is the value of the first case subform and (key ...) is the list of items at the front of each case clause. The case expression in the example above could be rewritten using cond as follows.

(let ([temp op])
  (cond
    [(memv temp '(add)) (apply-op ek + args)]
    [(memv temp '(sub)) (apply-op ek - args)]
    [(memv temp '(mul)) (apply-op ek * args)]
    [(memv temp '(div)) (apply-op ek / args)]
    [else (complain ek "invalid operator" op)])))

**Exercise 3.5.1**

Redefine complain in the calc example as an equivalent syntactic extension.

**Exercise 3.5.2**

In the calc example, the error continuation ek is passed along on each call to apply-op, complain, and do-calc. Move the definitions of apply-op, complain, and do-calc inward as far as necessary to eliminate the ek argument from the definitions and applications of these procedures.

**Exercise 3.5.3**

Eliminate the call/cc from calc and rewrite complain to raise an exception using assertion-violation.

**Exercise 3.5.4**

Extend calc to handle unary minus expressions, e.g.,

(calc '(minus (add 2 3))) ⇒ -5
and other operators of your choice.

Section 3.6. Libraries

At the end of the preceding section, we discussed a form of modularization that involves assigning a set of
top-level variables from within a let while keeping unpublished helpers local to the let. This form of
modularization has several drawbacks:

- It is unportable, because the behavior and even existence of an interactive top level is not guaranteed
  by the Revised Report.
- It requires assignments, which make the code appear somewhat awkward and may inhibit compiler
  analyses and optimizations.
- It does not support the publication of keyword bindings, since there is no analogue to set! for
  keywords.

An alternative that does not share these drawbacks is to create a library. A library exports a set of identifiers,
each defined within the library or imported from some other library. An exported identifier need not be bound
as a variable; it may be bound as a keyword instead.

The following library exports two identifiers: the variable gpa->grade and the keyword gpa. The variable
gpa->grade is bound to a procedure that takes a grade-point average (GPA), represented as a number, and
returns the corresponding letter grade, based on a four-point scale. The keyword gpa names a syntactic
extension whose subforms must all be letter grades and whose value is the GPA computed from those letter
grades.

(library (grades)
  (export gpa->grade gpa)
  (import (rnrs))

(define in-range?
  (lambda (x n y)
    (and (>= n x) (< n y))))

(define-syntax range-case
  (syntax-rules (- else)
    [(_ expr ((x - y) e1 e2 ...) ...
        [else ee1 ee2 ...])
      (let ([tmp expr])
        (cond
          [(in-range? x tmp y) e1 e2 ...]
          ...]
        [else ee1 ee2 ...]])
    [(_ expr ((x - y) e1 e2 ...) ...]
      (let ([tmp expr])
        (cond
          [(in-range? x tmp y) e1 e2 ...]
          ...)]))

(define letter->number
  (lambda (x)
    (case x
      [(a)  4.0]
      [(b)  3.0]
      [(c)  2.0]
      [(d)  1.0]  
      ...]))
(define gpa->grade
  (lambda (x)
    (range-case x
      [(0.0 - 0.5) 'f]
      [(0.5 - 1.5) 'd]
      [(1.5 - 2.5) 'c]
      [(2.5 - 3.5) 'b]
      [else 'a]]))

(define-syntax gpa
  (syntax-rules ()
    [(_ g1 g2 ...) (let ([ls (map letter->number '(g1 g2 ...))])
                    (/ (apply + ls) (length ls))))))

The name of the library is (grades). This may seem like a funny kind of name, but all library names are parenthesized. The library imports from the standard (rnrs) library, which contains most of the primitive and keyword bindings we have used in this chapter and the last, and everything we need to implement gpa->grade and gpa.

Along with gpa->grade and gpa, several other syntactic extensions and procedures are defined within the library, but none of the others are exported. The ones that aren't exported are simply helpers for the ones that are. Everything used within the library should be familiar, except for the apply procedure, which is described on page 107.

If your Scheme implementation supports import in the interactive top level, you can test the two exports as shown below.

(import (grades))
(gpa c a c b b)  \Rightarrow  2.8  
(gpa->grade 2.8)  \Rightarrow  b

Chapter 10 describes libraries in more detail and provides additional examples of their use.

Exercise 3.6.1

Modify gpa to handle "x" grades, which do not count in the grade-point average. Be careful to handle gracefully the situation where each grade is x.

(import (grades))
(gpa a x b c)  \Rightarrow  3.0

Exercise 3.6.2

Export from (grades) a new syntactic form, distribution, that takes a set of grades, like gpa, but returns a list of the form (((n g) ...)), where n is the number of times g appears in the set, with one entry for each g. Have distribution call an unexported procedure to do the actual work.

(import (grades))
(distribution a b a c c c a f b a)  \Rightarrow  ((4 a) (2 b) (3 c) (0 d) (1 f))
Exercise 3.6.3

Now read about output operations in Section 7.8 and define a new export, `histogram`, as a procedure that takes a `textual output port` and a distribution, such as might be produced by `distribution`, and prints a histogram in the style illustrated by the example below.

```scheme
(import (grades))
(histogram
  (current-output-port)
  (distribution a b a c c a c a f b a))
```

prints:
- a: ******
- b: **
- c: ***
- d: 
- f: *
Chapter 4. Procedures and Variable Bindings

Procedures and variable bindings are the fundamental building blocks of Scheme programs. This chapter describes the small set of syntactic forms whose primary purpose is to create procedures and manipulate variable bindings. It begins with the two most fundamental building blocks of Scheme programs: variable references and lambda expressions, and continues with descriptions of the variable binding and assignment forms such as define, letrec, let-values, and set!.

Various other forms that bind or assign variables for which the binding or assignment is not the primary purpose (such as named let) are found in Chapter 5.

Section 4.1. Variable References

**syntax:** variable
**returns:** the value of variable

Any identifier appearing as an expression in a program is a variable if a visible variable binding for the identifier exists, e.g., the identifier appears within the scope of a binding created by define, lambda, let, or some other variable-binding construct.

```
list ⇒ #<procedure>
(define x 'a)
(list x x) ⇒ (a a)
(let ([x 'b])
  (list x x)) ⇒ (b b)
(let ([let 'let]) let) ⇒ let
```

It is a syntax violation for an identifier reference to appear within a library form or top-level program if it is not bound as a variable, keyword, record name, or other entity. Since the scope of the definitions in a library, top-level program, lambda, or other local body is the entire body, it is not necessary for the definition of a variable to appear before its first reference appears, as long as the reference is not actually evaluated until the definition has been completed. So, for example, the reference to g within the definition of f below

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

is okay, but the reference to g in the definition of q below is not.

```
(define q (g 3))
(define g
  (lambda (x)
    (+ x x)))
```

Section 4.2. Lambda

**syntax:** (lambda formals body₁ body₂ ...)
**returns:** a procedure
**libraries:** (rnrs base), (rnrs)
The lambda syntactic form is used to create procedures. Any operation that creates a procedure or establishes local variable bindings is ultimately defined in terms of lambda or case-lambda.

The variables in forms are the formal parameters of the procedure, and the sequence of subforms body₁, body₂, ... is its body.

The body may begin with a sequence of definitions, in which case the bindings created by the definitions are local to the body. If definitions are present, the keyword bindings are used and discarded while expanding the body, and the body is expanded into a letrec* expression formed from the variable definitions and the remaining expressions, as described on page 292. The remainder of this description of lambda assumes that this transformation has taken place, if necessary, so that the body is a sequence of expressions without definitions.

When the procedure is created, the bindings of all variables occurring free within the body, excluding the formal parameters, are retained with the procedure. Subsequently, whenever the procedure is applied to a sequence of actual parameters, the formal parameters are bound to the actual parameters, the retained bindings are restored, and the body is evaluated.

Upon application, the formal parameters defined by forms are bound to the actual parameters as follows.

- If forms is a proper list of variables, e.g., (x y z), each variable is bound to the corresponding actual parameter. An exception with condition type &assertion is raised if too few or too many actual parameters are supplied.
- If forms is a single variable (not in a list), e.g., z, it is bound to a list of the actual parameters.
- If forms is an improper list of variables terminated by a variable, e.g., (x y . z), each variable but the last is bound to the corresponding actual parameter. The last variable is bound to a list of the remaining actual parameters. An exception with condition type &assertion is raised if too few actual parameters are supplied.

When the body is evaluated, the expressions in the body are evaluated in sequence, and the procedure returns the values of the last expression.

Procedures do not have a printed representation in the usual sense. Scheme systems print procedures in different ways; this book uses the notation #<procedure>.

\[
\text{(lambda (x) (+ x 3)) } \Rightarrow #\langle\text{procedure}\rangle \\
\text{((lambda (x) (+ x 3)) 7) } \Rightarrow 10 \\
\text{((lambda (x y) (* x (+ x y))) 7 13) } \Rightarrow 140 \\
\text{((lambda (f x) (f x x)) + 11) } \Rightarrow 22 \\
\text{((lambda () (+ 3 4))) } \Rightarrow 7 \\
\text{((lambda (x . y) (list x y)) 28 37) } \Rightarrow (28 (37)) \\
\text{((lambda (x . y) (list x y)) 28 37 47 28) } \Rightarrow (28 (37 47 28)) \\
\text{((lambda (x y . z) (list x y z)) 1 2 3 4) } \Rightarrow (1 2 (3 4)) \\
\text{((lambda x x) 7 13) } \Rightarrow (7 13)
\]

Section 4.3. Case-Lambda

A Scheme lambda expression always produces a procedure with a fixed number of arguments or with an indefinite number of arguments greater than or equal to a certain number. In particular,
(lambda (var₁ ... varₙ) body₁ body₂ ...) accepts exactly \( n \) arguments,

(lambda r body₁ body₂ ...) accepts zero or more arguments, and

(lambda (var₁ ... varₙ . r) body₁ body₂ ...) accepts \( n \) or more arguments.

\texttt{lambda} cannot directly produce, however, a procedure that accepts, say, either two or three arguments. In particular, procedures that accept optional arguments are not supported directly by \texttt{lambda}. The latter form of \texttt{lambda} shown above can be used, in conjunction with length checks and compositions of \texttt{car} and \texttt{cdr}, to implement procedures with optional arguments, though at the cost of clarity and efficiency.

The \texttt{case-lambda} syntactic form directly supports procedures with optional arguments as well as procedures with fixed or indefinite numbers of arguments. \texttt{case-lambda} is based on the \texttt{lambda*} syntactic form introduced in the article "A New Approach to Procedures with Variable Arity" [11].

**syntax:** (case-lambda clause ...)
**returns:** a procedure
**libraries:** (rnrs control), (rnrs)

A \texttt{case-lambda} expression consists of a set of clauses, each resembling a \texttt{lambda} expression. Each clause has the form below.

[formals body₁ body₂ ...]

The formal parameters of a clause are defined by \texttt{formals} in the same manner as for a \texttt{lambda} expression. The number of arguments accepted by the procedure value of a \texttt{case-lambda} expression is determined by the numbers of arguments accepted by the individual clauses.

When a procedure created with \texttt{case-lambda} is invoked, the clauses are considered in order. The first clause that accepts the given number of actual parameters is selected, the formal parameters defined by its \texttt{formals} are bound to the corresponding actual parameters, and the body is evaluated as described for \texttt{lambda} above. If \texttt{formals} in a clause is a proper list of identifiers, then the clause accepts exactly as many actual parameters as there are formal parameters (identifiers) in \texttt{formals}. As with a \texttt{lambda} \texttt{formals}, a \texttt{case-lambda} clause \texttt{formals} may be a single identifier, in which case the clause accepts any number of arguments, or an improper list of identifiers terminated by an identifier, in which case the clause accepts any number of arguments greater than or equal to the number of formal parameters excluding the terminating identifier. If no clause accepts the number of actual parameters supplied, an exception with condition type \&assertion is raised.

The following definition for \texttt{make-list} uses \texttt{case-lambda} to support an optional fill parameter.

(define make-list
  (case-lambda
   ([n) (make-list n #f)]
   [(n x)
    (do ([n n (- n 1)] [ls '()] (cons x ls)))
    ((zero? n) ls)))))

Section 4.3. Case-Lambda
The substring procedure may be extended with case-lambda to accept either no end index, in which case it defaults to the end of the string, or no start and end indices, in which case substring is equivalent to string-copy:

\[
(function \text{substring1} \\
\quad (\text{case-lambda} \\
\quad \quad [(s) \ (\text{substring1} \ s \ 0 \ (\text{string-length} \ s))] \\
\quad \quad [(s \ start) \ (\text{substring1} \ s \ start \ (\text{string-length} \ s))] \\
\quad \quad [(s \ start \ end) \ (\text{substring} \ s \ start \ end)])
\]

It is also possible to default the start index rather than the end index when only one index is supplied:

\[
(function \text{substring2} \\
\quad (\text{case-lambda} \\
\quad \quad [(s) \ (\text{substring2} \ s \ 0 \ (\text{string-length} \ s))] \\
\quad \quad [(s \ end) \ (\text{substring2} \ s \ 0 \ end)] \\
\quad \quad [(s \ start \ end) \ (\text{substring} \ s \ start \ end)])
\]

It is even possible to require that both or neither of the start and end indices be supplied, simply by leaving out the middle clause:

\[
(function \text{substring3} \\
\quad (\text{case-lambda} \\
\quad \quad [(s) \ (\text{substring3} \ s \ 0 \ (\text{string-length} \ s))] \\
\quad \quad [(s \ start \ end) \ (\text{substring} \ s \ start \ end)])
\]

Section 4.4. Local Binding

**Syntax:** \(\text{let}\ ((\var \ \text{expr}) \ ... \ \text{body}_1 \ \text{body}_2 \ ...)
\)

**Returns:** the values of the final body expression

**Libraries:** \((\text{rnrs base}), (\text{rnrs})\)

let establishes local variable bindings. Each variable \(\var\) is bound to the value of the corresponding expression \(\text{expr}\). The body of the let, in which the variables are bound, is the sequence of subforms \(\text{body}_1 \ \text{body}_2 \ ...\) and is processed and evaluated like a lambda body.

The forms \(\text{let, let*, letrec, and letrec*}\) (the others are described after let) are similar but serve slightly different purposes. With let, in contrast with let*, letrec, and letrec*, the expressions \(\text{expr} \ ...\) are all outside the scope of the variables \(\var \ ...\). Also, in contrast with let* and letrec*, no ordering is implied for the evaluation of the expressions \(\text{expr} \ ...\). They may be evaluated from left to right, from right to left, or in any other order at the discretion of the implementation. Use let whenever the values are independent of the variables and the order of evaluation is unimportant.

\[
(\text{let} \ ([(x \ (* \ 3.0 \ 3.0)) \ [y \ (* \ 4.0 \ 4.0)]]) \\
(\text{sqrt} \ (+ \ x \ y))) \Rightarrow 5.0
\]

\[
(\text{let} \ ([(x \ 'a) \ [y \ '(b \ c)]]) \\
(\text{cons} \ x \ y)) \Rightarrow (a \ b \ c)
\]

\[
(\text{let} \ ([(x \ 0) \ [y \ 1]]) \\
(\text{let} \ ([(x \ y) \ [y \ x]]) \\
(\text{list} \ x \ y))) \Rightarrow (1 \ 0)
\]

The following definition of let shows the typical derivation of let from lambda.
(define-syntax let
  (syntax-rules ()
    [(_ ((x e) ...) b1 b2 ...)
      ((lambda (x ...) b1 b2 ...) e ...)]))

Another form of let, named let, is described in Section 5.4, and a definition of the full let can be found on page 312.

**syntax:** (let* ((var expr ...) body₁ body₂ ...)
**returns:** the values of the final body expression
**libraries:** (rnrs base), (rnrs)

let* is similar to let except that the expressions expr ... are evaluated in sequence from left to right, and each of these expressions is within the scope of the variables to the left. Use let* when there is a linear dependency among the values or when the order of evaluation is important.

(let* ([x (* 5.0 5.0)]
       [y (- x (* 4.0 4.0))])
  (sqrt y)) ⇒ 3.0

(let ([x 0] [y 1])
  (let* ([x y] [y x])
    (list x y))) ⇒ (1 1)

Any let* expression may be converted to a set of nested let expressions. The following definition of let* demonstrates the typical transformation.

(define-syntax let*
  (syntax-rules ()
    [(_ () e1 e2 ...)
      (let () e1 e2 ...)]
    [(_ ((x1 v1) (x2 v2) ...) e1 e2 ...)
      (let ((x1 v1))
        (let* ((x2 v2) ...) e1 e2 ...))])

**syntax:** (letrec ((var expr ...) body₁ body₂ ...)
**returns:** the values of the final body expression
**libraries:** (rnrs base), (rnrs)

letrec is similar to let and let*, except that all of the expressions expr ... are within the scope of all of the variables var .... letrec allows the definition of mutually recursive procedures.

(letrec ([sum (lambda (x)
                   (if (zero? x)
                       0
                       (+ x (sum (- x 1)))))])
  (sum 5)) ⇒ 15

The order of evaluation of the expressions expr ... is unspecified, so a program must not evaluate a reference to any of the variables bound by the letrec expression before all of the values have been computed. (Occurrence of a variable within a lambda expression does not count as a reference, unless the resulting procedure is applied before all of the values have been computed.) If this restriction is violated, an exception with condition type &assertion is raised.
An `expr` should not return more than once. That is, it should not return both normally and via the invocation of a continuation obtained during its evaluation, and it should not return twice via two invocations of such a continuation. Implementations are not required to detect a violation of this restriction, but if they do, an exception with condition type `assertion` is raised.

Choose `letrec` over `let` or `let*` when there is a circular dependency among the variables and their values and when the order of evaluation is unimportant. Choose `letrec*` over `letrec` when there is a circular dependency and the bindings need to be evaluated from left to right.

A `letrec` expression of the form

```
(letrec ((var expr) ...) body₁ body₂ ...)
```

may be expressed in terms of `let` and `set!` as

```
(let ((var #f) ...)  
  (let ((temp expr) ...)  
    (set! var temp) ...  
    (let () body₁ body₂ ...)))
```

where `temp` ... are fresh variables, i.e., ones that do not already appear in the `letrec` expression, one for each `(var expr)` pair. The outer `let` expression establishes the variable bindings. The initial value given each variable is unimportant, so any value suffices in place of `#f`. The bindings are established first so that `expr` ... may contain occurrences of the variables, i.e., so that the expressions are computed within the scope of the variables. The middle `let` evaluates the values and binds them to the temporary variables, and the `set!` expressions assign each variable to the corresponding value. The inner `let` is present in case the body contains internal definitions.

A definition of `letrec` that uses this transformation is shown on page 310.

This transformation does not enforce the restriction that the `expr` expressions must not evaluate any references of or assignments to the variables. More elaborate transformations that enforce this restriction and actually produce more efficient code are possible [31].

**syntax:** `letrec* ((var expr) ...) body₁ body₂ ...`

**returns:** the values of the final body expression

**libraries:** `(rnrs base),(rnrs)`

`letrec*` is similar to `letrec`, except that `letrec*` evaluates `expr` ... in sequence from left to right. While programs must still not evaluate a reference to any `var` before the corresponding `expr` has been evaluated, references to `var` may be evaluated any time thereafter, including during the evaluation of the `expr` of any subsequent binding.

A `letrec*` expression of the form

```
(letrec* ((var expr) ...) body₁ body₂ ...)
```

may be expressed in terms of `let` and `set!` as

```
(let ((var #f) ...)  
  (set! var expr) ...  
  (let () body₁ body₂ ...))
```

The outer `let` expression creates the bindings, each assignment evaluates an expression and immediately sets
the corresponding variable to its value, in sequence, and the inner let evaluates the body. let is used in the latter case rather than begin since the body may include internal definitions as well as expressions.

(letrec* ([sum (lambda (x)
    (if (zero? x)
        0
        (+ x (sum (- x 1)))))]
    [f (lambda () (cons n n-sum))]
    [n 15]
    [n-sum (sum n)])
  (f)) ⇒ (15 . 120)

(letrec* ([f (lambda () (lambda () g))]
    [g (f)])
  (eq? (g) g)) ⇒ #t

(letrec* ([g (f)]
    [f (lambda () (lambda () g))])
  (eq? (g) g)) ⇒ exception: attempt to reference undefined variable f

Section 4.5. Multiple Values

syntax: (let-values ((formals expr) ...) body1 body2 ...)
syntax: (let*-values ((formals expr) ...) body1 body2 ...)
returns: the values of the final body expression
libraries: (rnrs base), (rnrs)

let-values is a convenient way to receive multiple values and bind them to variables. It is structured like let but permits an arbitrary formals list (like lambda) on each left-hand side. let*-values is similar but performs the bindings in left-to-right order, as with let*. An exception with condition type &assertion is raised if the number of values returned by an expr is not appropriate for the corresponding formals, as described in the entry for lambda above. A definition of let-values is given on page 310.

(let-values ([(a b) (values 1 2)] [(c (values 1 2 3)])
  (list a b c)) ⇒ (1 2 (1 2 3))

(let*-values ([(a b) (values 1 2)] [(a b) (values b a)])
  (list a b)) ⇒ (2 1)

Section 4.6. Variable Definitions

syntax: (define var expr)
syntax: (define var)
syntax: (define (var0 var1 ...) body1 body2 ...)
syntax: (define (var0 . varr) body1 body2 ...)
syntax: (define (var0 var1 var2 ... . varr) body1 body2 ...)
libraries: (rnrs base), (rnrs)

In the first form, define creates a new binding of var to the value of expr. The expr should not return more than once. That is, it should not return both normally and via the invocation of a continuation obtained during its evaluation, and it should not return twice via two invocations of such a continuation. Implementations are not required to detect a violation of this restriction, but if they do, an exception with condition type &assertion is raised.
The second form is equivalent to (define var unspecified), where unspecified is some unspecified value. The remaining are shorthand forms for binding variables to procedures; they are identical to the following definition in terms of lambda.

```
(define var 
  (lambda formals 
    body_1 body_2 ...))
```

where formals is (var_1 ...), var_r, or (var_1 var_2 ... var_r) for the third, fourth, and fifth define formats.

Definitions may appear at the front of a library body, anywhere among the forms of a top-level-program body, and at the front of a lambda or case-lambda body or the body of any form derived from lambda, e.g., let, or letrec*. Any body that begins with a sequence of definitions is transformed during macro expansion into a letrec* expression as described on page 292.

Syntax definitions may appear along with variable definitions wherever variable definitions may appear; see Chapter 8.

```
(define x 3)
x \Rightarrow 3
```

```
(define f 
  (lambda (x y) 
    (* (+ x y) 2)))
(f 5 4) \Rightarrow 18
```

```
(define (sum-of-squares x y) 
  (+ (* x x) (* y y)))
(sum-of-squares 3 4) \Rightarrow 25
```

```
(define f 
  (lambda (x) 
    (+ x 1)))
(let ([x 2]) 
  (define f 
    (lambda (y) 
      (+ y x)))
  (f 3)) \Rightarrow 5
(f 3) \Rightarrow 4
```

A set of definitions may be grouped by enclosing them in a begin form. Definitions grouped in this manner may appear wherever ordinary variable and syntax definitions may appear. They are treated as if written separately, i.e., without the enclosing begin form. This feature allows syntactic extensions to expand into groups of definitions.

```
(define-syntax multi-define-syntax 
  (syntax-rules () 
    [(_ (var expr) ...) 
      (begin 
        (define-syntax var expr) 
        ...)]))
(let () 
  (define plus
\[(\lambda (x\ y)\\n\hspace{1em}(if\ (zero?\ x)\ y\\n\hspace{1em}(plus\ (sub1\ x)\ (add1\ y)))))\\n\]

\[(\text{multi-define-syntex}\\n\hspace{1em}(\text{add1}\ (\text{syntax-rules}\ ()\ [(_\ e)\ (+\ e\ 1)]))\\n\hspace{1em}(\text{sub1}\ (\text{syntax-rules}\ ()\ [(_\ e)\ (-\ e\ 1)]))\\n\hspace{1em}(\text{plus}\ 7\ 8)) \Rightarrow 15\\n\]

Many implementations support an interactive "top level" in which variable and other definitions may be entered interactively or loaded from files. The behavior of these top-level definitions is outside the scope of the Revised\(^6\) Report, but as long as top-level variables are defined before any references or assignments to them are evaluated, the behavior is consistent across most implementations. So, for example, the reference to \(g\) in the top-level definition of \(f\) below is okay if \(g\) is not already defined, and \(g\) is assumed to name a variable to be defined at some later point.

\[(\text{define}\ f\\n\hspace{1em}(\lambda (x)\\n\hspace{2em}(g\ x)))\\n\]

If this is then followed by a definition of \(g\) before \(f\) is evaluated, the assumption that \(g\) would be defined as a variable is proven correct, and a call to \(f\) works as expected.

\[(\text{define}\ g\\n\hspace{1em}(\lambda (x)\\n\hspace{2em}(+\ x\ x)))\\n\hspace{1em}(f\ 3) \Rightarrow 6\\n\]

If \(g\) were defined instead as the keyword for a syntactic extension, the assumption that \(g\) would be bound as a variable is proven false, and if \(f\) is not redefined before it is invoked, the implementation is likely to raise an exception.

**Section 4.7. Assignment**

**Syntax:** \((\text{set!}\ \var\ \text{expr})\\n\)

**Returns:** unspecified

**Libraries:** (rnrs base), (rnrs)

\(\text{set!}\) does not establish a new binding for \(\var\) but rather alters the value of an existing binding. It first evaluates \(\text{expr}\), then assigns \(\var\) to the value of \(\text{expr}\). Any subsequent reference to \(\var\) within the scope of the altered binding evaluates to the new value.

Assignments are not employed as frequently in Scheme as in most other languages, but they are useful for implementing state changes.

\[(\text{define}\ \text{flip-flop}\\n\hspace{1em}(\text{let}\ ([\text{state}\ \#f])\\n\hspace{2em}(\lambda ()\\n\hspace{3em}(\text{set!}\ \text{state}\ (\text{not}\ \text{state}))\\n\hspace{2em}\text{state})))\\n\]

\[(\text{flip-flop}) \Rightarrow \#t\\n(\text{flip-flop}) \Rightarrow \#f\\n(\text{flip-flop}) \Rightarrow \#t\\n\]
Assignments are also useful for caching values. The example below uses a technique called memoization, in which a procedure records the values associated with old input values so it need not recompute them, to implement a fast version of the otherwise exponential doubly recursive definition of the Fibonacci function (see page 69).

```
(define memoize
  (lambda (proc)
    (let ([cache '()])
      (lambda (x)
        (cond
          [(assq x cache) => cdr]
          [else
           (let ([ans (proc x)])
             (set! cache (cons (cons x ans) cache))
             ans)]))))

(define fibonacci
  (memoize
    (lambda (n)
      (if (< n 2)
        1
        (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))

(fibonacci 100) => 573147844013817084101
```
Chapter 5. Control Operations

This chapter introduces the syntactic forms and procedures that serve as control structures for Scheme programs. The first section covers the most basic control structure, procedure application, and the remaining sections cover sequencing, conditional evaluation, recursion, mapping, continuations, delayed evaluation, multiple values, and evaluation of programs constructed at run time.

Section 5.1. Procedure Application

**syntax:** \((expr_0 \ expr_1 \ldots)\)
**returns:** values of applying the value of \(expr_0\) to the values of \(expr_1 \ldots\)

Procedure application is the most basic Scheme control structure. Any structured form without a syntax keyword in the first position is a procedure application. The expressions \(expr_0\) and \(expr_1 \ldots\) are evaluated; each should evaluate to a single value. After each of these expressions has been evaluated, the value of \(expr_0\) is applied to the values of \(expr_1 \ldots\). If \(expr_0\) does not evaluate to a procedure, or if the procedure does not accept the number of arguments provided, an exception with condition type \&assertion\ is raised.

The order in which the procedure and argument expressions are evaluated is unspecified. It may be left to right, right to left, or any other order. The evaluation is guaranteed to be sequential, however; whatever order is chosen, each expression is fully evaluated before evaluation of the next is started.

\((+ 3 4) \Rightarrow 7\)

\((\text{if} \ (\text{odd?} \ 3) \ + \ -) \ 6 \ 2) \Rightarrow 8\)

\((\text{lambda} \ (x) \ x) \ 5) \Rightarrow 5\)

\((\text{let} \ ((f \ (\text{lambda} \ (x) \ (+ \ x \ x))))\)
  \((f \ 8)) \Rightarrow 16\)

**procedure:** \((\text{apply} \ \text{procedure} \ \text{obj} \ldots \ \text{list})\)
**returns:** the values of applying \(\text{procedure}\) to \(\text{obj} \ldots\) and the elements of \(\text{list}\)

**libraries:** \((\text{rnrs base}), (\text{rnrs})\)

\(\text{apply}\) invokes \(\text{procedure}\), passing the first \(\text{obj}\) as the first argument, the second \(\text{obj}\) as the second argument, and so on for each object in \(\text{obj} \ldots\), and passing the elements of \(\text{list}\) in order as the remaining arguments. Thus, \(\text{procedure}\) is called with as many arguments as there are \(\text{objs}\) plus elements of \(\text{list}\).

\(\text{apply}\) is useful when some or all of the arguments to be passed to a procedure are in a list, since it frees the programmer from explicitly destructuring the list.

\((\text{apply} \ + \ '(4 \ 5)) \Rightarrow 9\)

\((\text{apply} \ \text{min} \ '(6 \ 8 \ 3 \ 2 \ 5)) \Rightarrow 2\)

\((\text{apply} \ \text{min} \ 5 \ 1 \ 3 \ '(6 \ 8 \ 3 \ 2 \ 5)) \Rightarrow 1\)

\((\text{apply} \ \text{vector} \ 'a \ 'b \ '(c \ d \ e)) \Rightarrow #\(\text{a b c d e}\)\)

\((\text{define} \ \text{first}\)
  \((\text{lambda} \ (\text{ls})\)
    \((\text{apply} \ (\text{lambda} \ (x \ . \ y) \ x) \ \text{ls})))\)

Chapter 5. Control Operations 87
(define rest
  (lambda (ls)
    (apply (lambda (x . y) y) ls)))

(first '(a b c d)) ⇒ a
(rest '(a b c d)) ⇒ (b c d)

(apply append
  '(1 2 3)
  '((a b) (c d e) (f))) ⇒ (1 2 3 a b c d e f)

Section 5.2. Sequencing

syntax: (begin expr₁ expr₂ ...)
returns: the values of the last subexpression
libraries: (rnrs base), (rnrs)

The expressions expr₁ expr₂ ... are evaluated in sequence from left to right. begin is used to sequence assignments, input/output, or other operations that cause side effects.

(define x 3)
(begin
  (set! x (+ x 1))
  (+ x x)) ⇒ 8

A begin form may contain zero or more definitions in place of the expressions expr₁ expr₂ ..., in which case it is considered to be a definition and may appear only where definitions are valid.

(let ()
  (begin (define x 3) (define y 4))
  (+ x y)) ⇒ 7

This form of begin is primarily used by syntactic extensions that must expand into multiple definitions. (See page 101.)

The bodies of many syntactic forms, including lambda, case-lambda, let, let*, letrec, and letrec*, as well as the result clauses of cond, case, and do, are treated as if they were inside an implicit begin; i.e., the expressions making up the body or result clause are executed in sequence, with the values of the last expression being returned.

(define swap-pair!
  (lambda (x)
    (let ([temp (car x)])
      (set-car! x (cdr x))
      (set-cdr! x temp)
      x)))

 swapping-pair! (cons 'a 'b)) ⇒ (b . a)

Section 5.3. Conditionals

syntax: (if test consequent alternative)
syntax: (if test consequent)
returns: the values of consequent or alternative depending on the value of test
libraries: (rnrs base), (rnrs)
The test, consequent, and alternative subforms must be expressions. If test evaluates to a true value (anything other than #f), consequent is evaluated and its values are returned. Otherwise, alternative is evaluated and its values are returned. With the second, "one-armed," form, which has no alternative, the result is unspecified if test evaluates to false.

(let ([ls '(a b c)])
  (if (null? ls)
      '()
      (cdr ls))) ⇒ (b c)

(let ([ls '()])
  (if (null? ls)
      '()
      (cdr ls))) ⇒ ()

(let ([abs
      (lambda (x)
        (if (< x 0)
            (- 0 x)
            x))])
  (abs -4)) ⇒ 4

(let ([x -4])
  (if (< x 0)
      (list 'minus (- 0 x))
      (list 'plus 4))) ⇒ (minus 4)

procedure: (not obj)
returns: #t if obj is false, #f otherwise
libraries: (rnrs base), (rnrs)

not is equivalent to (lambda (x) (if x #f #t)).

(not #f) ⇒ #t
(not #t) ⇒ #f
(not '()) ⇒ #f
(not (< 4 5)) ⇒ #f

syntax: (and expr ...)
returns: see below
libraries: (rnrs base), (rnrs)

If no subexpressions are present, the and form evaluates to #t. Otherwise, and evaluates each subexpression in sequence from left to right until only one subexpression remains or a subexpression returns #f. If one subexpression remains, it is evaluated and its values are returned. If a subexpression returns #f, and returns #f without evaluating the remaining subexpressions. A syntax definition of and appears on page 62.

(let ([x 3])
  (and (> x 2) (< x 4))) ⇒ #t

(let ([x 5])
  (and (> x 2) (< x 4))) ⇒ #f

(and #f '(a b) '(c d)) ⇒ #f
(and '(a b) '(c d) '(e f)) ⇒ (e f)

**syntax:** (or expr ...)
**returns:** see below
**libraries:** (rnrs base), (rnrs)

If no subexpressions are present, the or form evaluates to #f. Otherwise, or evaluates each subexpression in sequence from left to right until only one subexpression remains or a subexpression returns a value other than #f. If one subexpression remains, it is evaluated and its values are returned. If a subexpression returns a value other than #f, or returns that value without evaluating the remaining subexpressions. A syntax definition of or appears on page 63.

(let ([x 3])
  (or (< x 2) (> x 4))) ⇒ #f

(let ([x 5])
  (or (< x 2) (> x 4))) ⇒ #t

(or #f '(a b) '(c d)) ⇒ (a b)

**syntax:** (cond clause₁ clause₂ ...)
**returns:** see below
**libraries:** (rnrs base), (rnrs)

Each clause but the last must take one of the forms below.

(test)
(test expr₁ expr₂ ...)
(test => expr)

The last clause may be in any of the above forms, or it may be an "else clause" of the form

(else expr₁ expr₂ ...)

Each test is evaluated in order until one evaluates to a true value or until all of the tests have been evaluated. If the first clause whose test evaluates to a true value is in the first form given above, the value of test is returned.

If the first clause whose test evaluates to a true value is in the second form given above, the expressions expr₁ expr₂ ... are evaluated in sequence and the values of the last expression are returned.

If the first clause whose test evaluates to a true value is in the third form given above, the expression expr is evaluated. The value should be a procedure of one argument, which is applied to the value of test. The values of this application are returned.

If none of the tests evaluates to a true value and an else clause is present, the expressions expr₁ expr₂ ... of the else clause are evaluated in sequence and the values of the last expression are returned.

If none of the tests evaluates to a true value and no else clause is present, the value or values are unspecified.

See page 305 for a syntax definition of cond.
(let ([x 0])
  (cond
    [(< x 0) (list 'minus (abs x))]
    [(> x 0) (list 'plus x)]
    [else (list 'zero x)])  ⇒  (zero 0))

(define select
  (lambda (x)
    (cond
      [(not (symbol? x))]
      [(assq x '((a . 1) (b . 2) (c . 3))) => cdr]
      [else 0])))

(select 3)  ⇒  #t
(select 'b)  ⇒  2
(select 'e)  ⇒  0

These identifiers are auxiliary keywords for cond. Both also serve as auxiliary keywords for guard, and else also serves as an auxiliary keyword for case. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

.syntax: else
.syntax: =>
.libraries: (rnrs base), (rnrs exceptions), (rnrs)

For when, if test-expr evaluates to a true value, the expressions expr_1 expr_2 ... are evaluated in sequence, and the values of the last expression are returned. If test-expr evaluates to false, none of the other expressions are evaluated, and the value or values of when are unspecified.

For unless, if test-expr evaluates to false, the expressions expr_1 expr_2 ... are evaluated in sequence, and the values of the last expression are returned. If test-expr evaluates to a true value, none of the other expressions are evaluated, and the value or values of unless are unspecified.

A when or unless expression is usually clearer than the corresponding "one-armed" if expression.

(let ([x -4] [sign 'plus])
  (when (< x 0)
    (set! x (- 0 x))
    (set! sign 'minus))
  (list sign x))  ⇒  (minus 4)

(define check-pair
  (lambda (x)
    (unless (pair? x)
      (syntax-violation 'check-pair "invalid argument" x))
    x))

(check-pair '(a b c))  ⇒  (a b c)
when may be defined as follows:

(define-syntax when
  (syntax-rules ()
    [(_ e0 e1 e2 ...)
      (if e0 (begin e1 e2 ...))])))

unless may be defined as follows:

(define-syntax unless
  (syntax-rules ()
    [(_ e0 e1 e2 ...)
      (if (not e0) (begin e1 e2 ...))])))

or in terms of when as follows:

(define-syntax unless
  (syntax-rules ()
    [(_ e0 e1 e2 ...)
      (when (not e0) e1 e2 ...)]))

**syntax**: (case expr\(_0\) clause\(_1\) clause\(_2\) ...)

**returns**: see below

**libraries**: (rnrs base),(rnrs)

Each clause but the last must take the form

(((key ...) expr\(_1\) expr\(_2\) ...)

where each *key* is a datum distinct from the other keys. The last clause may be in the above form or it may be an *else* clause of the form

(\(\text{else}\) expr\(_1\) expr\(_2\) ...)

expr\(_0\) is evaluated and the result is compared (using eqv?) against the keys of each clause in order. If a clause containing a matching key is found, the expressions expr\(_1\) expr\(_2\) ... are evaluated in sequence and the values of the last expression are returned.

If none of the clauses contains a matching key and an *else* clause is present, the expressions expr\(_1\) expr\(_2\) ... of the *else* clause are evaluated in sequence and the values of the last expression are returned.

If none of the clauses contains a matching key and no *else* clause is present, the value or values are unspecified.

See page 306 for a syntax definition of case.

(let ([x 4] [y 5])
  (case (+ x y)
    [(1 3 5 7 9) 'odd]
    [(0 2 4 6 8) 'even]
    [else 'out-of-range])) ⇒ odd
Section 5.4. Recursion and Iteration

**syntax:** (let name ((var expr) ...) body₁ body₂ ...)  
**returns:** values of the final body expression  
**libraries:** (rnrs base), (rnrs)

This form of `let`, called *named let*, is a general-purpose iteration and recursion construct. It is similar to the more common form of `let` (see Section 4.4) in the binding of the variables `var` ... to the values of `expr` ... within the body `body₁ body₂` ..., which is processed and evaluated like a lambda body. In addition, the variable `name` is bound within the body to a procedure that may be called to recur or iterate; the arguments to the procedure become the new values of the variables `var` ... .

A named `let` expression of the form

```
(let name ((var expr) ...)  
 body₁ body₂ ...)
```

can be rewritten with `letrec` as follows.

```
(letrec ((name (lambda ((var ...) body₁ body₂ ...)))  
  expr ...)
```

A syntax definition of `let` that implements this transformation and handles unnamed `let` as well can be found on page 312.

The procedure `divisors` defined below uses named `let` to compute the nontrivial divisors of a nonnegative integer.

```
(define divisors  
  (lambda (n)  
    (let f ([i 2])  
      (cond  
        [(>= i n) '()]  
        [(integer? (/ n i)) (cons i (f (+ i 1)))]  
        [else (f (+ i 1))])))  
  )); (divisors 5) ⇒ ()  
  (divisors 32) ⇒ (2 4 8 16)
```

The version above is non-tail-recursive when a divisor is found and tail-recursive when a divisor is not found. The version below is fully tail-recursive. It builds up the list in reverse order, but this is easy to remedy, if desired, by reversing the list on exit.

```
(define divisors  
  (lambda (n)  
    (let f ([i 2] [ls '()])  
      (cond  
        [(>= i n) ls]  
        [(integer? (/ n i)) (f (+ i 1) (cons i ls))]  
        [else (f (+ i 1) ls))])))  
  )); syntax: (do (([var init update] ...) ([test result ...] expr ...))  
  returns: the values of the last result expression)
do allows a common restricted form of iteration to be expressed succinctly. The variables var... are bound initially to the values of init... and are rebound on each subsequent iteration to the values of update... The expressions test, update..., expr..., and result... are all within the scope of the bindings established for var....

On each step, the test expression test is evaluated. If the value of test is true, iteration ceases, the expressions result... are evaluated in sequence, and the values of the last expression are returned. If no result expressions are present, the value or values of the do expression are unspecified.

If the value of test is false, the expressions expr... are evaluated in sequence, the expressions update... are evaluated, new bindings for var... to the values of update... are created, and iteration continues.

The expressions expr... are evaluated only for effect and are often omitted entirely. Any update expression may be omitted, in which case the effect is the same as if the update were simply the corresponding var.

Although looping constructs in most languages require that the loop iterands be updated via assignment, do requires the loop iterands var... to be updated via rebinding. In fact, no side effects are involved in the evaluation of a do expression unless they are performed explicitly by its subexpressions.

See page 313 for a syntax definition of do.

The definitions of factorial and fibonacci below are straightforward translations of the tail-recursive named-let versions given in Section 3.2.

```
(define factorial
  (lambda (n)
    (do ([i n (- i 1)] [a 1 (* a i)])
      ((zero? i) a)))))

(factorial 10) ⇒ 3628800

(define fibonacci
  (lambda (n)
    (if (= n 0)
      0
      (do ([i n (- i 1)] [a1 1 (+ a1 a2)] [a2 0 a1])
        ((= i 1) a1)))))

(fibonacci 6) ⇒ 8
```

The definition of divisors below is similar to the tail-recursive definition of divisors given with the description of named let above.

```
(define divisors
  (lambda (n)
    (do ([i 2 (+ i 1)]
        [ls '()]
      (if (integer? (/ n i))
        (cons i ls)
        ls)))))
```
The definition of `scale-vector!` below, which scales each element of a vector \( v \) by a constant \( k \), demonstrates a nonempty `do` body.

```scheme
(define scale-vector!
  (lambda (v k)
    (let ([n (vector-length v)])
      (do ([i 0 (+ i 1)])
        ((= i n))
        (vector-set! v i (* (vector-ref v i) k))))))
```

```
(define vec (vector 1 2 3 4 5))
(scale-vector! vec 2)
vec ⇒ #(2 4 6 8 10)
```

### Section 5.5. Mapping and Folding

When a program must recur or iterate over the elements of a list, a mapping or folding operator is often more convenient. These operators abstract away from null checks and explicit recursion by applying a procedure to the elements of the list one by one. A few mapping operators are also available for vectors and strings.

**procedure:** `(map procedure list1 list2 ...)`  
**returns:** list of results  
**libraries:** `(rnrs base), (rnrs)`

`map` applies `procedure` to corresponding elements of the lists `list1 list2 ...` and returns a list of the resulting values. The lists `list1 list2 ...` must be of the same length. `procedure` should accept as many arguments as there are lists, should return a single value, and should not mutate the `list` arguments.

```scheme
(map abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)
```

```scheme
(map (lambda (x y) (* x y)) ' (1 2 3 4) ' (8 7 6 5)) ⇒ (8 14 18 20)
```

While the order in which the applications themselves occur is not specified, the order of the values in the output list is the same as that of the corresponding values in the input lists.

`map` might be defined as follows.

```scheme
(define map
  (lambda (f ls . more)
    (if (null? more)
      (let map1 ([ls ls])
        (if (null? ls)
          ()
          (cons (f (car ls))
            (map1 (cdr ls))))))
      (let map-more ([ls ls] [more more])
        (if (null? ls)
          ()
          (cons
            (apply f (car ls) (map car more))
            (map-more (cdr ls) more))))))
```
No error checking is done by this version of \texttt{map}; \(f\) is assumed to be a procedure and the other arguments are assumed to be proper lists of the same length. An interesting feature of this definition is that \texttt{map} uses itself to pull out the cars and cdrs of the list of input lists; this works because of the special treatment of the single-list case.

\textbf{procedure:} (for-each procedure list\_1 list\_2 ...)

\textbf{returns:} unspecified

\textbf{libraries:} (rnrs base), (rnrs)

\texttt{for-each} is similar to \texttt{map} except that \texttt{for-each} does not create and return a list of the resulting values, and \texttt{for-each} guarantees to perform the applications in sequence over the elements from left to right. \texttt{procedure} should accept as many arguments as there are lists and should not mutate the \texttt{list} arguments. \texttt{for-each} may be defined without error checks as follows.

\begin{verbatim}
(define for-each
  (lambda (f ls . more)
    (do ((ls ls (cdr ls)) [more more (map cdr more)])
        ((null? ls))
      (apply f (car ls) (map car more))))

(let (([same-count 0])
  (for-each
    (lambda (x y)
      (when (= x y)
        (set! same-count (+ same-count 1))))
    '(1 2 3 4 5 6)
    '(2 3 4 7 6))

  same-count) ⇒ 3)

procedure: (exists procedure list\_1 list\_2 ...)

\textbf{returns:} see below

\textbf{libraries:} (rnrs lists), (rnrs)

The lists \texttt{list\_1 list\_2 ...} must be of the same length. \texttt{procedure} should accept as many arguments as there are lists and should not mutate the \texttt{list} arguments. If the lists are empty, \texttt{exists} returns \#f. Otherwise, \texttt{exists} applies \texttt{procedure} to corresponding elements of the lists \texttt{list\_1 list\_2 ...} in sequence until either the lists each have only one element or \texttt{procedure} returns a true value \(t\). In the former case, \texttt{exists} tail-calls \texttt{procedure}, applying it to the remaining element of each list. In the latter case, \texttt{exists} returns \(t\).

\begin{verbatim}
(exists symbol? (1.0 #\a "hi" '())) ⇒ #f
(exists member (a b c)
  '((c b) (b a) (a c))) ⇒ (b a)
(exists (lambda (x y z) (= (+ x y) z))
  '(1 2 3 4)
  '(1.2 2.3 3.4 4.5)
  '(2.3 4.4 6.4 8.6)) ⇒ #t
\end{verbatim}

\texttt{exists} may be defined (somewhat inefficiently and without error checks) as follows:
(define exists
  (lambda (f ls . more)
    (and (not (null? ls))
      (let exists ([x (car ls)] [ls (cdr ls)] [more more])
        (if (null? ls)
            (apply f x (map car more))
            (or (apply f x (map car more))
                (exists (car ls) (cdr ls) (map cdr more))))))))

procedure: (for-all procedure list₁ list₂ ...)
returns: see below
libraries: (rnrs lists), (rnrs)

The lists list₁ list₂ ... must be of the same length. procedure should accept as many arguments as there are lists and should not mutate the list arguments. If the lists are empty, for-all returns #t. Otherwise, for-all applies procedure to corresponding elements of the lists list₁ list₂ ... in sequence until either the lists each have only one element left or procedure returns #f. In the former case, for-all tail-calls procedure, applying it to the remaining element of each list. In the latter case, for-all returns #f.

(for-all symbol? '(a b c d)) ⇒ #t
(for-all = '(1 2 3 4) '(1.0 2.0 3.0 4.0)) ⇒ #t
(for-all (lambda (x y z) (= (+ x y) z)) '(1 2 3 4) '(1.2 2.3 3.4 4.5) '(2.2 4.3 6.5 8.5)) ⇒ #f

for-all may be defined (somewhat inefficiently and without error checks) as follows:

(define for-all
  (lambda (f ls . more)
    (or (null? ls)
      (let for-all ([x (car ls)] [ls (cdr ls)] [more more])
        (if (null? ls)
            (apply f x (map car more))
            (and (apply f x (map car more))
                (for-all (car ls) (cdr ls) (map cdr more))))))))

procedure: (fold-left procedure obj list₁ list₂ ...)
returns: see below
libraries: (rnrs lists), (rnrs)

The list arguments should all have the same length. procedure should accept one more argument than the number of list arguments and return a single value. It should not mutate the list arguments.

fold-left returns obj if the list arguments are empty. If they are not empty, fold-left applies procedure to obj and the cars of list₁ list₂ ..., then recurs with the value returned by procedure in place of obj and the cdr of each list in place of the list.

(fold-left cons () '(1 2 3 4)) ⇒ ((((() . 1) . 2) . 3) . 4)
(fold-left
  (lambda (a x) (+ a (* x x)))
  0 '(1 2 3 4 5)) ⇒ 55

(fold-left
  (lambda (a . args) (append args a))
  '(question)
  '(that not to)
  '(is to be)
  '(the be: or)) ⇒ (to be or not to be: that is the question)

procedure: (fold-right procedure obj list₁ list₂ ...)
returns: see below
libraries: (rnrs lists), (rnrs)

The list arguments should all have the same length. procedure should accept one more argument than the number of list arguments and return a single value. It should not mutate the list arguments.

fold-right returns obj if the list arguments are empty. If they are not empty, fold-right recurs with the cdr of each list replacing the list, then applies procedure to the cars of list₁ list₂ ... and the result returned by the recursion.

(fold-right cons '() '(1 2 3 4)) ⇒ (1 2 3 4)

(fold-right
  (lambda (x a) (+ a (* x x)))
  0 '(1 2 3 4 5)) ⇒ 55

(fold-right
  (lambda (x y a) (cons* x y a))
  '(parting is such sweet sorrow
    (with apologies))
  '(gotta go see ya tomorrow
    (is sweet gotta see tomorrow))

procedure: (vector-map procedure vector₁ vector₂ ...) returns: vector of results
libraries: (rnrs base), (rnrs)

vector-map applies procedure to corresponding elements of vector₁ vector₂ ... and returns a vector of the resulting values. The vectors vector₁ vector₂ ... must be of the same length, and procedure should accept as many arguments as there are vectors and return a single value.

(vector-map abs '#(1 -2 3 -4 5 -6)) ⇒ #(1 2 3 4 5 6)
(vector-map (lambda (x y) (* x y))
  '#(1 2 3 4)
  '#(8 7 6 5)) ⇒ #(8 14 18 20)

While the order in which the applications themselves occur is not specified, the order of the values in the output vector is the same as that of the corresponding values in the input vectors.

procedure: (vector-for-each procedure vector₁ vector₂ ...) returns: unspecified
libraries: (rnrs base), (rnrs)
vector-for-each is similar to vector-map except that vector-for-each does not create and return a vector of the resulting values, and vector-for-each guarantees to perform the applications in sequence over the elements from left to right.

\[
\text{(let ([same-count 0])}
\text{(vector-for-each}
\text{  (lambda (x y)
\text{    (when (= x y)
\text{      (set! same-count (+ same-count 1))))
\text{  #'(1 2 3 4 5 6)
\text{  #'(2 3 4 7 6))}
\text{same-count) ⇒ 3)}
\]

procedure: (string-for-each procedure string₁ string₂ ...)  
returns: unspecified  
libraries: (rnrs base), (rnrs)

string-for-each is similar to for-each and vector-for-each except that the inputs are strings rather than lists or vectors.

\[
\text{(let ([ls '()])}
\text{(string-for-each}
\text{  (lambda r (set! ls (cons r ls)))
\text{  "abcd"
\text{  "===="
\text{  "1234")}
\text{  (map list->string (reverse ls))) ⇒ ("a=1" "b=2" "c=3" "d=4")}
\]

Section 5.6. Continuations

Continuations in Scheme are procedures that represent the remainder of a computation from a given point in the computation. They may be obtained with call-with-current-continuation, which can be abbreviated to call/cc.

procedure: (call/cc procedure)  
procedure: (call-with-current-continuation procedure)  
returns: see below  
libraries: (rnrs base), (rnrs)

These procedures are the same. The shorter name is often used for the obvious reason that it requires fewer keystrokes to type.

call/cc obtains its continuation and passes it to procedure, which should accept one argument. The continuation itself is represented by a procedure. Each time this procedure is applied to zero or more values, it returns the values to the continuation of the call/cc application. That is, when the continuation procedure is called, it returns its arguments as the values of the application of call/cc.

If procedure returns normally when passed the continuation procedure, the values returned by call/cc are the values returned by procedure.

Continuations allow the implementation of nonlocal exits, backtracking [14, 29], coroutines [16], and multitasking [10, 32].

The example below illustrates the use of a continuation to perform a nonlocal exit from a loop.
(define member)
  (lambda (x ls)
    (call/cc
      (lambda (break)
        (do ((ls ls (cdr ls)))
            ((null? ls) #f)
            (when (equal? x (car ls))
              (break ls)))))))

(member 'd '(a b c)) \Rightarrow #f
(member 'b '(a b c)) \Rightarrow (b c)

Additional examples are given in Sections 3.3 and 12.11.

The current continuation is typically represented internally as a stack of procedure activation records, and obtaining the continuation involves encapsulating the stack within a procedural object. Since an encapsulated stack has indefinite extent, some mechanism must be used to preserve the stack contents indefinitely. This can be done with surprising ease and efficiency and with no impact on programs that do not use continuations [17].

**procedure**: (dynamic-wind in body out)
**returns**: values resulting from the application of body
**libraries**: (rnrs base), (rnrs)

**dynamic-wind** offers "protection" from continuation invocation. It is useful for performing tasks that must be performed whenever control enters or leaves body, either normally or by continuation application.

The three arguments in, body, and out must be procedures and should accept zero arguments, i.e., they should be thunks. Before applying body, and each time body is entered subsequently by the application of a continuation created within body, the in thunk is applied. Upon normal exit from body and each time body is exited by the application of a continuation created outside body, the out thunk is applied.

Thus, it is guaranteed that in is invoked at least once. In addition, if body ever returns, out is invoked at least once.

The following example demonstrates the use of dynamic-wind to be sure that an input port is closed after processing, regardless of whether the processing completes normally.

(let ([p (open-input-file "input-file")])
  (dynamic-wind
    (lambda () #f)
    (lambda () (process p))
    (lambda () (close-port p))))

Common Lisp provides a similar facility (unwind-protect) for protection from nonlocal exits. This is often sufficient. unwind-protect provides only the equivalent to out, however, since Common Lisp does not support fully general continuations. Here is how unwind-protect might be specified with dynamic-wind.

(define-scheme-syntactic-unwind-protect
  (syntax-rules ()
    [(_ body cleanup ...) (dynamic-wind
        (lambda () #f)
Some Scheme implementations support a controlled form of assignment known as fluid binding, in which a variable takes on a temporary value during a given computation and reverts to the old value after the computation has completed. The syntactic form fluid-let defined below in terms of dynamic-wind permits the fluid binding of a single variable \( x \) to the value of an expression \( e \) within a the body \( b_1 b_2 \ldots \).

\[
\begin{align*}
\text{(lambda () body)} \\
\text{(lambda () cleanup ...))})
\end{align*}
\]

\[
\begin{align*}
\text{((call/cc} \\
\text{(let ([(x 'a)])} \\
\text{(lambda (k)} \\
\text{(unwind-protect} \\
\text{(k (lambda () x))} \\
\text{(set! x 'b)))))}) \Rightarrow b
\end{align*}
\]

Implementations that support fluid-let typically extend it to allow an indefinite number of \((x e)\) pairs, as with let.

If no continuations are invoked within the body of a fluid-let, the behavior is the same as if the variable were simply assigned the new value on entry and assigned the old value on return.

\[
\begin{align*}
\text{(let ([(x 3)])} \\
\text{(+ (fluid-let ([(x 5)])} \\
\text{x) \Rightarrow 8)
\end{align*}
\]

A fluid-bound variable also reverts to the old value if a continuation created outside of the fluid-let is invoked.

\[
\begin{align*}
\text{(let ([(x 'a)])} \\
\text{(let ([(f (lambda () x)])} \\
\text{(cons (call/cc} \\
\text{(lambda (k)} \\
\text{fluid-let ([(x 'b)])} \\
\text{(k (f))))))} \\
\text{(f)))) \Rightarrow (b . a)
\end{align*}
\]

If control has left a fluid-let body, either normally or by the invocation of a continuation, and control reenters the body by the invocation of a continuation, the temporary value of the fluid-bound variable is reinstated. Furthermore, any changes to the temporary value are maintained and reflected upon reentry.

\[
\begin{align*}
\text{(define reenter #f)} \\
\text{(define x 0)} \\
\text{(fluid-let ([(x 1)])} \\
\text{(call/cc (lambda (k) (set! reenter k))} \\
\text{(set! x (+ x 1)))
\end{align*}
\]
A library showing how `dynamic-wind` might be implemented were it not already built in is given below. In addition to defining `dynamic-wind`, the code defines a version of `call/cc` that does its part to support `dynamic-wind`.

(library (dynamic-wind)
  (export dynamic-wind call/cc
      (rename (call/cc call-with-current-continuation)))
  (import (rename (except (rnrs) dynamic-wind) (call/cc rnrs:call/cc)))

(define winders '())

(define common-tail
  (lambda (x y)
    (let ([lx (length x)] [ly (length y)])
      (do ([x (if (> lx ly) (list-tail x (- lx ly)) x) (cdr x)]
          [y (if (> ly lx) (list-tail y (- ly lx)) y) (cdr y)])
          ((eq? x y) x)))))

(define do-wind
  (lambda (new)
    (let ([tail (common-tail new winders)])
      (let f ([ls winders])
        (if (not (eq? ls tail))
          (begin
            (set! winders (cdr ls))
            ((cdar ls))
            (f (cdr ls)))))
      (let f ([ls new])
        (if (not (eq? ls tail))
          (begin
            (f (cdr ls))
            ((caar ls))
            (set! winders ls))))))))

(define call/cc
  (lambda (f)
    (rnrs:call/cc
      (lambda (k)
        (f (let ([save winders])
            (lambda (x)
              (unless (eq? save winders) (do-wind save))
              (k x))))))))

(define dynamic-wind
  (lambda (in body out)
    (in)
    (set! winders (cons (cons in out) winders))
    (let-values ([ans* (body)])
      ...))
Together, *dynamic-wind* and *call/cc* manage a list of *winders*. A winder is a pair of *in* and *out* thunks established by a call to *dynamic-wind*. Whenever *dynamic-wind* is invoked, the *in* thunk is invoked, a new winder containing the *in* and *out* thunks is placed on the winders list, the *body* thunk is invoked, the winder is removed from the winders list, and the *out* thunk is invoked. This ordering ensures that the winder is on the winders list only when control has passed through *in* and not yet entered *out*. Whenever a continuation is obtained, the winders list is saved, and whenever the continuation is invoked, the saved winders list is reinstated. During reinstatement, the *out* thunk of each winder on the current winders list that is not also on the saved winders list is invoked, followed by the *in* thunk of each winder on the saved winders list that is not also on the current winders list. The winders list is updated incrementally, again to ensure that a winder is on the current winders list only if control has passed through its *in* thunk and not entered its *out* thunk.

The test `(not (eq? save winders))` performed in *call/cc* is not strictly necessary but makes invoking a continuation less costly whenever the saved winders list is the same as the current winders list.

### Section 5.7. Delayed Evaluation

The syntactic form *delay* and the procedure *force* may be used in combination to implement lazy evaluation. An expression subject to lazy evaluation is not evaluated until its value is required and, once evaluated, is never reevaluated.

**Syntax:**

```
(delay expr)
```

**Returns:** a promise

**Procedure:**

```
(force promise)
```

**Returns:** result of forcing *promise*

**Libraries:** (rnrs r5rs)

The first time a promise created by *delay* is *forced* (with *force*), it evaluates *expr*, "remembering" the resulting value. Thereafter, each time the promise is forced, it returns the remembered value instead of reevaluating *expr*.

*delay* and *force* are typically used only in the absence of side effects, e.g., assignments, so that the order of evaluation is unimportant.

The benefit of using *delay* and *force* is that some amount of computation might be avoided altogether if it is delayed until absolutely required. Delayed evaluation may be used to construct conceptually infinite lists, or *streams*. The example below shows how a stream abstraction may be built with *delay* and *force*. A stream is a promise that, when forced, returns a pair whose cdr is a stream.

```
(define stream-car
  (lambda (s)
    (car (force s))))

(define stream-cdr
  (lambda (s)
    (cdr (force s))))

(define counters
  (let next ([n 1])
    (delay (cons n (next (+ n 1)))))))
```
(stream-car counters) ⇒ 1

(stream-car (stream-cdr counters)) ⇒ 2

(define stream-add
  (lambda (s1 s2)
    (delay (cons
            (+ (stream-car s1) (stream-car s2))
            (stream-add (stream-cdr s1) (stream-cdr s2))))))

(define even-counters
  (stream-add counters counters))

(stream-car even-counters) ⇒ 2

(stream-car (stream-cdr even-counters)) ⇒ 4

delay may be defined by

(define-syntax delay
  (syntax-rules ()
    [(_ expr) (make-promise (lambda () expr))]))

where make-promise might be defined as follows.

(define make-promise
  (lambda (p)
    (let ([vals #f] [set? #f])
      (lambda ()
        (unless set?
          (let ([x (p)])
            (unless set?
              (set! val x)
              (set! set? #t)))
          val))))

With this definition of delay, force simply invokes the promise to force evaluation or to retrieve the saved value.

(define force
  (lambda (promise)
    (promise)))

The second test of the variable set? in make-promise is necessary in the event that, as a result of applying p, the promise is recursively forced. Since a promise must always return the same value, the result of the first application of p to complete is returned.

Whether delay and force handle multiple return values is unspecified; the implementation given above does not, but the following version does, with the help of call-with-values and apply.

(define make-promise
  (lambda (p)
    (let ([vals #f] [set? #f])
      (lambda ()
        (let-values ((vals (p)))
          (unless set?
            (set! vals x)
            (set! set? #t)))
          vals)))
Neither implementation is quite right, since force must raise an exception with condition type &assertion if its argument is not a promise. Since distinguishing procedures created by make-promise from other procedures is impossible, force cannot do so reliably. The following reimplementation of make-promise and force represents promises as records of the type promise to allow force to make the required check.

(define-record-type promise
  (fields (immutable p) (mutable vals) (mutable set?))
  (protocol (lambda (new) (lambda (p) (new p #f #f)))))

(define force
  (lambda (promise)
    (unless (promise? promise)
      (assertion-violation 'promise "invalid argument" promise))
    (unless (promise-set? promise)
      (call-with-values (promise-p promise)
        (lambda x
          (unless (promise-set? promise)
            (promise-vals-set! promise x)
            (promise-set?-set! promise #t))))))
    (apply values (promise-vals promise)))))

Section 5.8. Multiple Values

While all Scheme primitives and most user-defined procedures return exactly one value, some programming problems are best solved by returning zero values, more than one value, or even a variable number of values. For example, a procedure that partitions a list of values into two sublists needs to return two values. While it is possible for the producer of multiple values to package them into a data structure and for the consumer to extract them, it is often cleaner to use the built-in multiple-values interface. This interface consists of two procedures: values and call-with-values. The former produces multiple values and the latter links procedures that produce multiple-value values with procedures that consume them.

procedure: (values obj ...)
returns: obj ...
libraries: (rnrs base),(rnrs)

The procedure values accepts any number of arguments and simply passes (returns) the arguments to its continuation.
(values) ⇒

(values 1) ⇒ 1

(values 1 2 3) ⇒ 1
2
3

(define head&tail
  (lambda (ls)
    (values (car ls) (cdr ls)))))

(head&tail '(a b c)) ⇒ a
  (b c)

procedure: (call-with-values producer consumer)
returns: see below
libraries: (rnrs base), (rnrs)

producer and consumer must be procedures. call-with-values applies consumer to the values returned by invoking producer without arguments.

(call-with-values
  (lambda () (values 'bond 'james))
  (lambda (x y) (cons y x)))
⇒ (james . bond)

(call-with-values values list) ⇒ '()

In the second example, values itself serves as the producer. It receives no arguments and thus returns no values. list is thus applied to no arguments and so returns the empty list.

The procedure dxdy defined below computes the change in x and y coordinates for a pair of points whose coordinates are represented by (x . y) pairs.

(define dxdy
  (lambda (p1 p2)
    (values (- (car p2) (car p1))
     (- (cdr p2) (cdr p1))))))

(dxdy '(0 . 0) '(0 . 5)) ⇒ 0
  5

dxdy can be used to compute the length and slope of a segment represented by two endpoints.

(define segment-length
  (lambda (p1 p2)
    (call-with-values
      (lambda () (dxdy p1 p2))
      (lambda (dx dy) (sqrt (+ (* dx dx) (* dy dy)))))))

(define segment-slope
  (lambda (p1 p2)
    (call-with-values
      (lambda () (dxdy p1 p2))
We can of course combine these to form one procedure that returns two values.

\[
\text{(define describe-segment}
  \text{ (lambda (p1 p2)
       (call-with-values
         (lambda () (dxdy p1 p2))
         (lambda (dx dy)
           (values
             (sqrt (+ (* dx dx) (* dy dy)))
             (/ dy dx))))))}
\]

\[
\text{(describe-segment '}(1 . 4) '}(4 . 8)) \Rightarrow 5
\]
\[
\text{\qquad} \Rightarrow 4/3
\]

The example below employs multiple values to divide a list nondestructively into two sublists of alternating elements.

\[
\text{(define split}
  \text{ (lambda (ls)
       (if (or (null? ls) (null? (cdr ls)))
         (values ls '())
         (call-with-values
           (lambda () (split (cddr ls)))
           (lambda (odds evens)
             (values (cons (car ls) odds)
                   (cons (cadr ls) evens))))))})
\]

\[
\text{(split '(a b c d e f)) \Rightarrow (a c e)
\]
\[
\text{\qquad (b d f)}
\]

At each level of recursion, the procedure split returns two values: a list of the odd-numbered elements from the argument list and a list of the even-numbered elements.

The continuation of a call to values need not be one established by a call to call-with-values, nor must only values be used to return to a continuation established by call-with-values. In particular, \(\text{(values e)}\) and \(e\) are equivalent expressions. For example:

\[
\text{(+ (values 2) 4) \Rightarrow 6}
\]
\[
\text{(if (values #t) 1 2) \Rightarrow 1}
\]

\[
\text{(call-with-values}
  \text{ (lambda () 4)
         (lambda (x) x)) \Rightarrow 4}
\]

Similarly, values may be used to pass any number of values to a continuation that ignores the values, as in the following.

\[
\text{(begin (values 1 2 3) 4) \Rightarrow 4}
\]
Because a continuation may accept zero or more than one value, continuations obtained via \texttt{call/cc} may accept zero or more than one argument.

\begin{verbatim}
(call-with-values
 (lambda ()
   (call/cc (lambda (k) (k 2 3))))
 (lambda (x y) (list x y))) ⇒ (2 3)
\end{verbatim}

The behavior is unspecified when a continuation expecting exactly one value receives zero values or more than one value. For example, the behavior of each of the following expressions is unspecified. Some implementations raise an exception, while others silently suppress additional values or supply defaults for missing values.

\begin{verbatim}
(if (values 1 2) 'x 'y)
(+ (values) 5)
\end{verbatim}

Programs that wish to force extra values to be ignored in particular contexts can do so easily by calling \texttt{call-with-values} explicitly. A syntactic form, which we might call \texttt{first}, can be defined to abstract the discarding of more than one value when only one is desired.

\begin{verbatim}
(define-syntax first
 (syntax-rules ()
   [(_ expr)
     (call-with-values
      (lambda () expr)
      (lambda (x . y) x))])))
\end{verbatim}

\begin{verbatim}
(if (first (values #t #f)) 'a 'b) ⇒ a
\end{verbatim}

Since implementations are required to raise an exception with condition type \texttt{assertion} if a procedure does not accept the number of arguments passed to it, each of the following raises an exception.

\begin{verbatim}
(call-with-values
 (lambda () (values 2 3 4))
 (lambda (x y) x))
\end{verbatim}

\begin{verbatim}
(call-with-values
 (lambda () (call/cc (lambda (k) (k 0))))
 (lambda (x y) x))
\end{verbatim}

Since \texttt{producer} is most often a \texttt{lambda} expression, it is often convenient to use a syntactic extension that suppresses the lambda expression in the interest of readability.

\begin{verbatim}
(define-syntax with-values
 (syntax-rules ()
   [(_ expr consumer)
     (call-with-values (lambda () expr) consumer)]))
\end{verbatim}

\begin{verbatim}
(with-values (values 1 2) list) ⇒ (1 2)
(with-values (split '(1 2 3 4))
 (lambda (odds evens)
   evens)) ⇒ (2 4)
\end{verbatim}
If the consumer is also a lambda expression, the multiple-value variants of let and let* described in Section 4.5 are usually even more convenient.

(let-values ([[(odds evens) (split '(1 2 3 4))]])
  evens) ⇒ (2 4)

(let-values ([[ls (values 'a 'b 'c)]])
  ls) ⇒ (a b c)

Many standard syntactic forms and procedures pass along multiple values. Most of these are “automatic,” in the sense that nothing special must be done by the implementation to make this happen. The usual expansion of let into a direct lambda call automatically propagates multiple values produced by the body of the let. Other operators must be coded specially to pass along multiple values. The call-with-port procedure (page 7.6), for example, calls its procedure argument, then closes the port argument before returning the procedure’s values, so it must save the values temporarily. This is easily accomplished via let-values, apply, and values:

(define call-with-port
  (lambda (port proc)
    (let-values ([[val* (proc port)]])
      (close-port port)
      (apply values val*)))))

If this seems like too much overhead when a single value is returned, the code can use call-with-values and case-lambda to handle the single-value case more efficiently:

(define call-with-port
  (lambda (port proc)
    (call-with-values (lambda () (proc port))
      (case-lambda
        [((val) (close-port port) val)
        [(val* (close-port port) (apply values val*))])))))

The definitions of values and call-with-values (and concomitant redefinition of call/cc) in the library below demonstrate that the multiple-return-values interface could be implemented in Scheme if it were not already built in. No error checking can be done, however, for the case in which more than one value is returned to a single-value context, such as the test part of an if expression.

(library (mrvs)
  (export call-with-values values call/cc
    (rename (call/cc call-with-current-continuation)))
  (import
    (rename
      (except (rnrs) values call-with-values)
      (call/cc rnrs:call/cc)))

  (define magic (cons 'multiple 'values))

  (define magic?
    (lambda (x)
      (and (pair? x) (eq? (car x) magic)))

  (define call/cc
    (lambda (p)
(rnrs:call/cc
 (lambda (k)
   (p (lambda args
        (k (apply values args)))))))

(define values
 (lambda args
   (if (and (not (null? args)) (null? (cdr args)))
       (car args)
       (cons magic args)))))

(define call-with-values
 (lambda (producer consumer)
   (let ([x (producer)])
     (if (magic? x)
         (apply consumer (cdr x))
         (consumer x))))))

Multiple values can be implemented more efficiently [2], but this code serves to illustrate the meanings of the operators and may be used to provide multiple values in older, nonstandard implementations that do not support them.

Section 5.9. Eval

Scheme's `eval` procedure allows programmers to write programs that construct and evaluate other programs. This ability to do run-time meta programming should not be overused but is handy when needed.

**procedure:** `(eval obj environment)`  
**returns:** values of the Scheme expression represented by `obj` in `environment`  
**libraries:** `(rnrs eval)`

If `obj` does not represent a syntactically valid expression, `eval` raises an exception with condition type `&syntax`. The environments returned by `environment`, `scheme-report-environment`, and `null-environment` are immutable. Thus, `eval` also raises an exception with condition type `&syntax` if an assignment to any of the variables in the environment appears within the expression.

```
(define cons 'not-cons)
(eval '(let ([x 3]) (cons x 4)) (environment '(rnrs))) ⇒ (3 . 4)

(define lambda 'not-lambda)
(eval '(lambda (x) x) (environment '(rnrs))) ⇒ #<procedure>

(eval '(cons 3 4) (environment)) ⇒ exception
```

**procedure:** `(environment import-spec ...)`  
**returns:** an environment  
**libraries:** `(rnrs eval)`

`environment` returns an environment formed from the combined bindings of the given import specifiers. Each `import-spec` must be an s-expression representing a valid import specifier (see Chapter 10).

```
(define env (environment '(rnrs) '(prefix (rnrs lists) $)))
(eval '($cons* 3 4 (* 5 8)) env) ⇒ (3 4 . 40)
```
procedure: (null-environment version)

procedure: (scheme-report-environment version)

returns: an R5RS compatibility environment

libraries: (rnrs r5rs)

version must be the exact integer 5.

null-environment returns an environment containing bindings for the keywords whose meanings are defined by the Revised\textsuperscript{5} Report on Scheme, along with bindings for the auxiliary keywords else, =>, ..., and _.

scheme-report-environment returns an environment containing the same keyword bindings as the environment returned by null-environment along with bindings for the variables whose meanings are defined by the Revised\textsuperscript{5} Report on Scheme, except those not defined by the Revised\textsuperscript{6} Report: load, interaction-environment, transcript-on, transcript-off, and char-ready?.

The bindings for each of the identifiers in the environments returned by these procedures are those of the corresponding Revised\textsuperscript{6} Report library, so this does not provide full backward compatibility, even if the excepted identifier bindings are not used.
Chapter 6. Operations on Objects

This chapter describes the operations on objects, including lists, numbers, characters, strings, vectors, bytevectors, symbols, booleans, hashtables, and enumerations. The first section covers constant objects and quotation. The second section describes generic equivalence predicates for comparing two objects and predicates for determining the type of an object. Later sections describe procedures that deal primarily with one of the object types mentioned above. There is no section treating operations on procedures, since the only operation defined specifically for procedures is application, and this is described in Chapter 5. Operations on ports are covered in the more general discussion of input and output in Chapter 7. A mechanism for defining new data types is described in Chapter 9.

Section 6.1. Constants and Quotation

**syntax:** constant  
**returns:** constant

*constant* is any self-evaluating constant, i.e., a number, boolean, character, string, or bytevector. Constants are immutable; see the note in the description of *quote* below.

3.2 \Rightarrow 3.2  
#f \Rightarrow #f  
#\c \Rightarrow #\c  
"hi" \Rightarrow "hi"  
#vu8(3 4 5) \Rightarrow #vu8(3 4 5)

**syntax:** (quote obj)  
**returns:** obj  
**libraries:** (rnrs base), (rnrs)

'(obj is equivalent to (quote obj). The abbreviated form is converted into the longer form by the Scheme reader (see read).

*quote* inhibits the normal evaluation rule for *obj*, allowing *obj* to be employed as data. Although any Scheme object may be quoted, quotation is not necessary for self-evaluating constants, i.e., numbers, booleans, characters, strings, and bytevectors.

Quoted and self-evaluating constants are immutable. That is, programs should not alter a constant via *set-car!*,*string-set!*,*cons*, etc., and implementations are permitted to raise an exception with condition type *&assertion* if such an alteration is attempted. If an attempt to alter an immutable object is undetected, the behavior of the program is unspecified. An implementation may choose to share storage among different constants to save space.

(+ 2 3) \Rightarrow 5  
'(+ 2 3) \Rightarrow (+ 2 3)  
(quote (+ 2 3)) \Rightarrow (+ 2 3)  
'a \Rightarrow a  
'cons \Rightarrow cons  
'() \Rightarrow ()  
'7 \Rightarrow 7

**syntax:** (quasiquote obj ...)  
**syntax:** `obj  
**syntax:** (unquote obj ...)

Chapter 6. Operations on Objects
syntax: ,obj
syntax: (unquote-splicing obj ...)
syntax: ,@obj
returns: see below
libraries: (rnrs base), (rnrs)

`obj` is equivalent to `(quasiquote obj)`. ,obj is equivalent to `(unquote obj)`, and ,@obj is equivalent to `(unquote-splicing obj)`. The abbreviated forms are converted into the longer forms by the Scheme reader (see read).

quasiquote is similar to quote, but it allows parts of the quoted text to be "unquoted." Within a quasiquote expression, unquote and unquote-splicing subforms are evaluated, and everything else is quoted, i.e., left unevaluated. The value of each unquote subform is inserted into the output in place of the unquote form, while the value of each unquote-splicing subform is spliced into the surrounding list or vector structure. unquote and unquote-splicing are valid only within quasiquote expressions.

quasiquote expressions may be nested, with each quasiquote introducing a new level of quotation and each unquote or unquote-splicing taking away a level of quotation. An expression nested within `n` quasiquote expressions must be within `n` unquote or unquote-splicing expressions to be evaluated.

`(+ 2 3) ⇒ (+ 2 3)
(+ 2 ,(* 3 4)) ⇒ (+ 2 12)
(a b ,(+ 2 3) c) d ⇒ (a b (5 c) d)
(a b ,reverse `(c d e)) f g ⇒ (a b (e d c) f g)
(let ([a 1] [b 2])
  ,(a . ,b)) ⇒ (1 . 2)
(+ ,@((cdr '(* 2 3)))) ⇒ (+ 2 3)
(a b ,@((reverse `(c d e)) f g)) ⇒ (a b (e d c) f g)
(let ([a 1] [b 2])
  ,(a ,@b)) ⇒ (1 . 2)
#,(@,(list 1 2 3)) ⇒ #(1 2 3)
',(cons 'a 'b) ⇒ ',(cons 'a 'b)
',(cons 'a 'b) ⇒ '(a . b)

unquote and unquote-splicing forms with zero or more than one subform are valid only in splicing (list or vector) contexts. (unquote obj ...) is equivalent to (unquote obj) ..., and (unquote-splicing obj ...) is equivalent to (unquote-splicing obj) .... These forms are primarily useful as intermediate forms in the output of the quasiquote expander. They support certain useful nested quasiquotation idioms [3], such as ,@,@, which has the effect of a doubly indirect splicing when used within a doubly nested and doubly evaluated quasiquote expression.

(a (unquote b)) ⇒ (a b)
(a (unquote (+ 3 3)) b) ⇒ (a 6 b)
(a (unquote-splicing (+ 3 3)) (° 3 3)) b) ⇒ (a 6 9 b)
(let ([x '(m n)]) `(a ,@,@x f)) ⇒ `(a (unquote-splicing m n) f)
(let ([x '(m n)])
  (eval `(let ([m '(b c)] [n '(d e)]) `(a ,@,@x f))
    (environment '(rnrs)))) ⇒ (a b c d e f)
unquote and unquote-splicing are auxiliary keywords for quasiquote. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

Section 6.2. Generic Equivalence and Type Predicates

This section describes the basic Scheme predicates (procedures returning one of the boolean values \#t or \#f) for determining the type of an object or the equivalence of two objects. The equivalence predicates eq?, eqv?, and equal? are discussed first, followed by the type predicates.

**procedure:** (eq? obj1 obj2)
**returns:** \#t if obj1 and obj2 are identical, \#f otherwise
**libraries:** (rnrs base), (rnrs)

In most Scheme systems, two objects are considered identical if they are represented internally by the same pointer value and distinct (not identical) if they are represented internally by different pointer values, although other criteria, such as time-stamping, are possible.

Although the particular rules for object identity vary somewhat from system to system, the following rules always hold.

- Two objects of different types (booleans, the empty list, pairs, numbers, characters, strings, vectors, symbols, and procedures) are distinct.
- Two objects of the same type with different contents or values are distinct.
- The boolean object \#t is identical to itself wherever it appears, and \#f is identical to itself wherever it appears, but \#t and \#f are distinct.
- The empty list () is identical to itself wherever it appears.
- Two symbols are identical if and only if they have the same name (by string=?).
- A constant pair, vector, string, or bytevector is identical to itself, as is a pair, vector, string, or bytevector created by an application of cons, vector, string, make-bytevector, etc. Two pairs, vectors, strings, or bytevectors created by different applications of cons, vector, string, make-bytevector, etc., are distinct. One consequence is that cons, for example, may be used to create a unique object distinct from all other objects.
- Two procedures that may behave differently are distinct. A procedure created by an evaluation of a lambda expression is identical to itself. Two procedures created by the same lambda expression at different times, or by similar lambda expressions, may or may not be distinct.

eq? cannot be used to compare numbers and characters reliably. Although every inexact number is distinct from every exact number, two exact numbers, two inexact numbers, or two characters with the same value may or may not be identical.

Since constant objects are immutable, i.e., programs should not modify them via vector-set!, set-car!, or any other structure mutation operation, all or portions of different quoted constants or self-evaluating literals may be represented internally by the same object. Thus, eq? may return \#t when applied to equal parts of different immutable constants.

**eq? is most often used to compare symbols or to check for pointer equivalence of allocated objects, e.g., pairs, vectors, or record instances.**

\[
\begin{align*}
& (\text{eq?} \ 'a \ 3) \Rightarrow \ \#f \\
& (\text{eq?} \ \#t \ 't) \Rightarrow \ \#f \\
& (\text{eq?} \ "abc" \ 'abc) \Rightarrow \ \#f \\
& (\text{eq?} \ "hi" \ '(hi)) \Rightarrow \ \#f \\
& (\text{eq?} \ \#f \ '()) \Rightarrow \ \#f
\end{align*}
\]
The Scheme Programming Language, 4th Edition

Section 6.2. Generic Equivalence and Type Predicates

(eq? 9/2 7/2) ⇒ #f
(eq? 3.4 53344) ⇒ #f
(eq? 3 3.0) ⇒ #f
(eq? 1/3 #i1/3) ⇒ #f

(eq? 9/2 9/2) ⇒ unspecified
(eq? 3.4 (+ 3.0 .4)) ⇒ unspecified
(let ([x (* 12345678987654321 2)])
  (eq? x x)) ⇒ unspecified

(eq? \a \b) ⇒ #f
(eq? \a \a) ⇒ unspecified
(let ([x (string-ref "hi" 0)])
  (eq? x x)) ⇒ unspecified

(eq? #t #t) ⇒ #t
(eq? #f #f) ⇒ #t
(eq? #t #f) ⇒ #f
(eq? (null? '()) #t) ⇒ #t
(eq? (null? '(a)) #f) ⇒ #t

(eq? (cdr '(a)) '()) ⇒ #t

(eq? 'a 'a) ⇒ #t
(eq? 'a 'b) ⇒ #f
(eq? 'a (string->symbol "a")) ⇒ #t

(eq? '(a) '(b)) ⇒ #f
(eq? '(a)'(a)) ⇒ unspecified
(let ([x '(a . b)]) (eq? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
  (eq? x x)) ⇒ #t
(eq? (cons 'a 'b) (cons 'a 'b)) ⇒ #f

(eq? "abc" "cba") ⇒ #f
(eq? "abc" "abc") ⇒ unspecified
(let ([x "hi"] (eq? x x)) ⇒ #t
(let ([x (string #\h #\i)]) (eq? x x)) ⇒ #t
(eq? (string #\h #\i)
  (string #\h #\i)) ⇒ #f

(eq? '#vu8(l) '#vu8(l)) ⇒ unspecified
(eq? '#vu8(l) '#vu8(2)) ⇒ #f
(let ([x (make-bytevector 10 0)])
  (eq? x x)) ⇒ #t
(let ([x (make-bytevector 10 0)])
  (eq? x (make-bytevector 10 0))) ⇒ #f

(eq? '#(a) '#(b)) ⇒ #f
(eq? '#(a) '#(a)) ⇒ unspecified
(let ([x '#(a)]) (eq? x x)) ⇒ #t
(let ([x (vector 'a)])
  (eq? x x)) ⇒ #t
(eq? (vector 'a) (vector 'a)) ⇒ #f
(eq? car car) ⇒ #t
(eq? car cdr) ⇒ #f
(let ([f (lambda (x) x))]
  (eq? f f)) ⇒ #t
(let ([f (lambda () (lambda (x) x))])
  (eq? (f) (f))) ⇒ unspecified
(eq? (lambda (x) x) (lambda (y) y)) ⇒ unspecified

(let ([f (lambda (x)
            (lambda ()
                (set! x (+ x 1))
                x))])
  (eq? (f 0) (f 0))) ⇒ #f

procedure: (eqv? obj1 obj2)
returns: #t if obj1 and obj2 are equivalent, #f otherwise
libraries: (rnrs base), (rnrs)

eqv? is similar to eq? except eqv? is guaranteed to return #t for two characters that are considered equal by char=? and two numbers that are (a) considered equal by = and (b) cannot be distinguished by any other operation besides eq? and eqv?. A consequence of (b) is that (eqv? -0.0 +0.0) is #f even though (= -0.0 +0.0) is #t in systems that distinguish -0.0 and +0.0, such as those based on IEEE floating-point arithmetic. This is because operations such as / can expose the difference:

(/ 1.0 -0.0) ⇒ -inf.0
(/ 1.0 +0.0) ⇒ +inf.0

Similarly, although 3.0 and 3.0+0.0i are considered numerically equal, they are not considered equivalent by eqv? if -0.0 and 0.0 have different representations.

(= 3.0+0.0i 3.0) ⇒ #t
(eqv? 3.0+0.0i 3.0) ⇒ #f

The boolean value returned by eqv? is not specified when the arguments are NaNs.

(eqv? +nan.0 (/ 0.0 0.0)) ⇒ unspecified

eqv? is less implementation-dependent but generally more expensive than eq?.

(eqv? 'a 3) ⇒ #f
(eqv? #t 't) ⇒ #f
(eqv? "abc" 'abc) ⇒ #f
(eqv? "hi" '(hi)) ⇒ #f
(eqv? #f '()) ⇒ #f

(eqv? 9/2 7/2) ⇒ #f
(eqv? 3.4 53344) ⇒ #f
(eqv? 3 3.0) ⇒ #f
(eqv? 1/3 #i1/3) ⇒ #f

(eqv? 9/2 9/2) ⇒ #t
(eqv? 3.4 (+ 3.0 .4)) ⇒ #t
(let ([x (* 12345678987654321 2)])
The Scheme Programming Language, 4th Edition

(eqv? x x) ⇒ #t

(eqv? \a \b) ⇒ #f
(eqv? \a \a) ⇒ #t
(let ([x (string-ref "hi" 0)])
  (eqv? x x)) ⇒ #t

(eqv? #t #t) ⇒ #t
(eqv? #f #f) ⇒ #t
(eqv? #t #f) ⇒ #f
(eqv? (null? '()) #t) ⇒ #t
(eqv? (null? '(a)) #f) ⇒ #t

(eqv? (cdr '(a)) '()) ⇒ #t

(eqv? 'a 'a) ⇒ #t
(eqv? 'a 'b) ⇒ #f
(eqv? 'a (string->symbol "a")) ⇒ #t

(eqv? '(a) '(b)) ⇒ #f
(eqv? '(a) '(a)) ⇒ unspecified
(let ([x '(a . b)]) (eqv? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
  (eqv? x x)) ⇒ #t
(eqv? (cons 'a 'b) (cons 'a 'b)) ⇒ #f

(eqv? "abc" "cba") ⇒ #f
(eqv? "abc" "abc") ⇒ unspecified
(let ([x "hi"])
  (eqv? x x)) ⇒ #t
(let ([x (string #\h #\i)]) (eqv? x x)) ⇒ #t
(eqv? (string #\h #\i)
  (string #\h #\i)) ⇒ #f

(eqv? '#vu8(1) '#vu8(1)) ⇒ unspecified
(eqv? '#vu8(1) '#vu8(2)) ⇒ #f
(let ([x (make-bytevector 10 0)])
  (eqv? x x)) ⇒ #t
(let ([x (make-bytevector 10 0)])
  (eqv? x (make-bytevector 10 0))) ⇒ #f

(eqv? '#(a) '#(b)) ⇒ #f
(eqv? '#(a) '#(a)) ⇒ unspecified
(let ([x '#(a)]) (eqv? x x)) ⇒ #t
(let ([x (vector 'a)])
  (eqv? x x)) ⇒ #t
(eqv? (vector 'a) (vector 'a)) ⇒ #f

(eqv? car car) ⇒ #t
(eqv? car cdr) ⇒ #f
(let ([f (lambda (x) x)])
  (eqv? f f)) ⇒ #t
(let ([f (lambda () (lambda (x) x))])
  (eqv? (f) (f))) ⇒ unspecified
(eqv? (lambda (x) x) (lambda (y) y)) ⇒ unspecified
(let ([f (lambda (x)
  (lambda ()
    (set! x (+ x 1))
    x))])
  (eqv? (f 0) (f 0))) ⇒ #f

**procedure**: (equal? obj₁ obj₂)

**returns**: #t if obj₁ and obj₂ have the same structure and contents, #f otherwise

**libraries**: (rnrs base), (rnrs)

Two objects are equal if they are equivalent according to eqv?, strings that are string=?, bytevectors that are bytevector=?, pairs whose cars and cdrs are equal, or vectors of the same length whose corresponding elements are equal.

equal? is required to terminate even for cyclic arguments and return #t "if and only if the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees" [24]. In essence, two values are equivalent, in the sense of equal?, if the structure of the two objects cannot be distinguished by any composition of pair and vector accessors along with the eqv?, string=?, and bytevector=? procedures for comparing data at the leaves.

Implementing equal? efficiently is tricky [1], and even with a good implementation, it is likely to be more expensive than either eqv? or eq?.

(equal? 'a 3) ⇒ #f
(equal? #t 't) ⇒ #f
(equal? "abc" 'abc) ⇒ #f
(equal? "hi" '(hi)) ⇒ #f
(equal? #f '()) ⇒ #f

(equal? 9/2 7/2) ⇒ #f
(equal? 3.4 53344) ⇒ #f
(equal? 3 3.0) ⇒ #f
(equal? 1/3 #i1/3) ⇒ #f

(equal? 9/2 9/2) ⇒ #t
(equal? 3.4 (+ 3.0 .4)) ⇒ #t
(let ([x (* 12345678987654321 2)])
  (equal? x x)) ⇒ #t

(equal? #\a #\b) ⇒ #f
(equal? #\a #\a) ⇒ #t
(let ([x (string-ref "hi" 0)])
  (equal? x x)) ⇒ #t

(equal? #t #t) ⇒ #t
(equal? #f #f) ⇒ #t
(equal? #t #f) ⇒ #f
(equal? (null? '()) #t) ⇒ #t
(equal? (null? '(a)) #f) ⇒ #t

(equal? (cdr '(a)) '()) ⇒ #t

(equal? 'a 'a) ⇒ #t
(equal? 'a 'b) ⇒ #f
(equal? 'a (string->symbol "a")) ⇒ #t

(equal? '(a) '(b)) ⇒ #f
(equal? '(a) '(a)) ⇒ #t
(let ((x '(a . b))) (equal? x x)) ⇒ #t
(let ((x (cons 'a 'b)))
  (equal? x x)) ⇒ #t
(equal? (cons 'a 'b) (cons 'a 'b)) ⇒ #t

(equal? "abc" "cba") ⇒ #f
(equal? "abc" "abc") ⇒ #t
(let ((x "hi")) (equal? x x)) ⇒ #t
(let ((x (string #\h #\i))) (equal? x x)) ⇒ #t
(equal? (string #\h #\i)
        (string #\h #\i)) ⇒ #t

(equal? '#vu8(1) '#vu8(1)) ⇒ #t
(equal? '#vu8(1) '#vu8(2)) ⇒ #f
(let ((x (make-bytevector 10 0)))
  (equal? x x)) ⇒ #t
(let ((x (make-bytevector 10 0)))
  (equal? x (make-bytevector 10 0))) ⇒ #t

(equal? '#(a) '#(b)) ⇒ #f
(equal? '#(a) '#(a)) ⇒ #t
(let ((x '#(a))) (equal? x x)) ⇒ #t
(let ((x (vector 'a)))
  (equal? x x)) ⇒ #t
(equal? (vector 'a) (vector 'a)) ⇒ #t

(equal? car car) ⇒ #t
(equal? car cdr) ⇒ #f
(let ((f (lambda (x) x)))
  (equal? f f)) ⇒ #t
(let ((f (lambda () (lambda (x) x))))
  (equal? (f) (f))) ⇒ unspecified
(equal? (lambda (x) x) (lambda (y) y)) ⇒ unspecified

(let ((f (lambda (x)
            (lambda ()
              (set! x (+ x 1))
              x)))))
(equal? (f 0) (f 0))) ⇒ #f

(equal?
  (let ((x (cons 'x 'x)))
    (set-car! x x)
    (set-cdr! x x)
    x)
  (let ((x (cons 'x 'x)))
    (set-car! x x)
    (set-cdr! x x)
    (cons x x))) ⇒ #t
**procedure**: (boolean? obj)
**returns**: #t if obj is either #t or #f, #f otherwise
**libraries**: (rnrs base), (rnrs)

boolean? is equivalent to (lambda (x) (or (eq? x #t) (eq? x #f))).

(boolean? #t) ⇒ #t  
(boolean? #f) ⇒ #t  
(or (boolean? 't) (boolean? '())) ⇒ #f

**procedure**: (null? obj)
**returns**: #t if obj is the empty list, #f otherwise
**libraries**: (rnrs base), (rnrs)

null? is equivalent to (lambda (x) (eq? x '())).

(null? '()) ⇒ #t  
(null? '(a)) ⇒ #f  
(null? (cdr '(a))) ⇒ #t  
(null? 3) ⇒ #f  
(null? #f) ⇒ #f

**procedure**: (pair? obj)
**returns**: #t if obj is a pair, #f otherwise
**libraries**: (rnrs base), (rnrs)

(pair? '(a b c)) ⇒ #t  
(pair? '(3 . 4)) ⇒ #t  
(pair? '()) ⇒ #f  
(pair? '#(a b)) ⇒ #f  
(pair? 3) ⇒ #f

**procedure**: (number? obj)
**returns**: #t if obj is a number object, #f otherwise

**procedure**: (complex? obj)
**returns**: #t if obj is a complex number object, #f otherwise

**procedure**: (real? obj)
**returns**: #t if obj is a real number object, #f otherwise

**procedure**: (rational? obj)
**returns**: #t if obj is a rational number object, #f otherwise

**procedure**: (integer? obj)
**returns**: #t if obj is an integer object, #f otherwise
**libraries**: (rnrs base), (rnrs)

These predicates form a hierarchy: any integer is rational, any rational is real, any real is complex, and any complex is numeric. Most implementations do not provide internal representations for irrational numbers, so all real numbers are typically rational as well.

The real?, rational?, and integer? predicates do not recognize as real, rational, or integer complex numbers with inexact zero imaginary parts.

(integer? 1901) ⇒ #t  
(rational? 1901) ⇒ #t  
(real? 1901) ⇒ #t
(complex? 1901) ⇒ #t
(number? 1901) ⇒ #t

(integer? -3.0) ⇒ #t
(rational? -3.0) ⇒ #t
(real? -3.0) ⇒ #t
(complex? -3.0) ⇒ #t
(number? -3.0) ⇒ #t

(integer? 7+0i) ⇒ #t
(rational? 7+0i) ⇒ #t
(real? 7+0i) ⇒ #t
(complex? 7+0i) ⇒ #t
(number? 7+0i) ⇒ #t

(integer? -2/3) ⇒ #f
(rational? -2/3) ⇒ #t
(real? -2/3) ⇒ #t
(complex? -2/3) ⇒ #t
(number? -2/3) ⇒ #t

(integer? -2.345) ⇒ #f
(rational? -2.345) ⇒ #t
(real? -2.345) ⇒ #t
(complex? -2.345) ⇒ #t
(number? -2.345) ⇒ #t

(integer? 7.0+0.0i) ⇒ #f
(rational? 7.0+0.0i) ⇒ #f
(real? 7.0+0.0i) ⇒ #f
(complex? 7.0+0.0i) ⇒ #t
(number? 7.0+0.0i) ⇒ #t

(integer? 3.2-2.01i) ⇒ #f
(rational? 3.2-2.01i) ⇒ #f
(real? 3.2-2.01i) ⇒ #f
(complex? 3.2-2.01i) ⇒ #t
(number? 3.2-2.01i) ⇒ #t

(integer? 'a) ⇒ #f
(rational? '(a b c)) ⇒ #f
(real? "3") ⇒ #f
(complex? '#(1 2)) ⇒ #f
(number? #\a) ⇒ #f

procedure: (real-valued? obj)
returns: #t if obj is a real number, #f otherwise

procedure: (rational-valued? obj)
returns: #t if obj is a rational number, #f otherwise

procedure: (integer-valued? obj)
returns: #t if obj is an integer, #f otherwise

libraries: (rnrs base), (rnrs)

These predicates are similar to real?, rational?, and integer?, but treat as real, rational, or integral
complex numbers with inexact zero imaginary parts.

(integer-valued? 1901) ⇒ #t
(rational-valued? 1901) ⇒ #t
(real-valued? 1901) ⇒ #t

(integer-valued? -3.0) ⇒ #t
(rational-valued? -3.0) ⇒ #t
(real-valued? -3.0) ⇒ #t

(integer-valued? 7+0i) ⇒ #t
(rational-valued? 7+0i) ⇒ #t
(real-valued? 7+0i) ⇒ #t

(integer-valued? -2/3) ⇒ #f
(rational-valued? -2/3) ⇒ #t
(real-valued? -2/3) ⇒ #t

(integer-valued? -2.345) ⇒ #f
(rational-valued? -2.345) ⇒ #t
(real-valued? -2.345) ⇒ #t

(integer-valued? 7.0+0.0i) ⇒ #t
(rational-valued? 7.0+0.0i) ⇒ #t
(real-valued? 7.0+0.0i) ⇒ #t

(integer-valued? 3.2-2.01i) ⇒ #f
(rational-valued? 3.2-2.01i) ⇒ #f
(real-valued? 3.2-2.01i) ⇒ #f

As with real?, rational?, and integer?, these predicates return #f for all non-numeric values.

(integer-valued? 'a) ⇒ #f
(rational-valued? "(a b c)) ⇒ #f
(real-valued? "3") ⇒ #f

procedure: (char? obj)
returns: #t if obj is a character, #f otherwise
libraries: (rnrs base), (rnrs)

(char? 'a) ⇒ #f
(char? 97) ⇒ #f
(char? \a) ⇒ #t
(char? "a") ⇒ #f
(char? (string-ref (make-string 1) 0)) ⇒ #t

procedure: (string? obj)
returns: #t if obj is a string, #f otherwise
libraries: (rnrs base), (rnrs)

(string? "hi") ⇒ #t
(string? 'hi) ⇒ #f
(string? \h) ⇒ #f
procedure: (vector? obj)
returns: #t if obj is a vector, #f otherwise
libraries: (rnrs base), (rnrs)

(vector? '(())) ⇒ #t
(vector? '((a b c))) ⇒ #t
(vector? (vector 'a 'b 'c)) ⇒ #t
(vector? '()) ⇒ #f
(vector? '(a b c)) ⇒ #f
(vector? "abc") ⇒ #f

procedure: (symbol? obj)
returns: #t if obj is a symbol, #f otherwise
libraries: (rnrs base), (rnrs)

(symbol? 't) ⇒ #t
(symbol? "t") ⇒ #f
(symbol? '(t)) ⇒ #f
(symbol? '\t) ⇒ #f
(symbol? 3) ⇒ #f
(symbol? #t) ⇒ #f

procedure: (procedure? obj)
returns: #t if obj is a procedure, #f otherwise
libraries: (rnrs base), (rnrs)

(procedure? car) ⇒ #t
(procedure? 'car) ⇒ #f
(procedure? (lambda (x) x)) ⇒ #t
(call/cc procedure?) ⇒ #t

procedure: (bytevector? obj)
returns: #t if obj is a bytevector, #f otherwise
libraries: (rnrs bytevectors), (rnrs)

(bytevector? #vu8()) ⇒ #t
(bytevector? '(())) ⇒ #f
(bytevector? "abc") ⇒ #f

procedure: (hashtable? obj)
returns: #t if obj is a hashtable, #f otherwise
libraries: (rnrs hashtables), (rnrs)

(hashtable? (make-eq-hashtable)) ⇒ #t
(hashtable? '(not a hash table)) ⇒ #f

Section 6.3. Lists and Pairs

The pair, or cons cell, is the most fundamental of Scheme's structured object types. The most common use for pairs is to build lists, which are ordered sequences of pairs linked one to the next by the cdr field. The elements of the list occupy the car fields of the pairs. The cdr of the last pair in a proper list is the empty list, (); the cdr of the last pair in an improper list can be anything other than ().
Pairs may be used to construct binary trees. Each pair in the tree structure is an internal node of the binary tree; its car and cdr are the children of the node.

Proper lists are printed as sequences of objects separated by whitespace and enclosed in parentheses. Matching pairs of brackets ([ ]) may be used in place of parentheses. For example, (1 2 3) and (a [nested list]) are proper lists. The empty list is written as ()

Improper lists and trees require a slightly more complex syntax. A single pair is written as two objects separated by whitespace and a dot, e.g., (a . b). This is referred to as *dotted-pair notation*. Improper lists and trees are also written in dotted-pair notation; the dot appears wherever necessary, e.g., (1 2 3 . 4) or ((1 . 2) . 3). Proper lists may be written in dotted-pair notation as well. For example, (1 2 3) may be written as (1 . (2 . (3 . ()))).

It is possible to create a circular list or a cyclic graph by destructively altering the car or cdr field of a pair, using `set-car!` or `set-cdr!`. Such lists are not considered proper lists.

Procedures that accept a *list* argument are required to detect that the list is improper only to the extent that they actually traverse the list far enough either (a) to attempt to operate on a non-list tail or (b) to loop indefinitely due to a circularity. For example, `member` need not detect that a list is improper if it actually finds the element being sought, and `list-ref` need never detect circularities, because its recursion is bounded by the index argument.

**procedure:** `(cons obj₁ obj₂)`  
**returns:** a new pair whose car and cdr are `obj₁` and `obj₂`  
**libraries:** `(rnrs base),(rnrs)`

`cons` is the pair constructor procedure. `obj₁` becomes the car and `obj₂` becomes the cdr of the new pair.

- `(cons 'a '()) ⇒ (a)`  
- `(cons 'a '(b c)) ⇒ (a b c)`  
- `(cons 3 4) ⇒ (3 . 4)`

**procedure:** `(car pair)`  
**returns:** the car of `pair`  
**libraries:** `(rnrs base),(rnrs)`

The empty list is not a pair, so the argument must not be the empty list.

- `(car '(a)) ⇒ a`  
- `(car '(a b c)) ⇒ a`  
- `(car (cons 3 4)) ⇒ 3`

**procedure:** `(cdr pair)`  
**returns:** the cdr of `pair`  
**libraries:** `(rnrs base),(rnrs)`

The empty list is not a pair, so the argument must not be the empty list.

- `(cdr '(a)) ⇒ ()`  
- `(cdr '(a b c)) ⇒ (b c)`  
- `(cdr (cons 3 4)) ⇒ 4`

**procedure:** `(set-car! pair obj)`  
**returns:** unspecified
set-car! changes the car of pair to obj.

(let ([x (list 'a 'b 'c)])
  (set-car! x 1)
  x) ⇒ (1 b c)

procedure: (set-cdr! pair obj)
returns: unspecified
libraries: (rnrs mutable-pairs)

set-cdr! changes the cdr of pair to obj.

(let ([x (list 'a 'b 'c)])
  (set-cdr! x 1)
  x) ⇒ (a . 1)

procedure: (caar pair)
procedure: (cadr pair):
procedure: (cddddr pair)
returns: the caar, cadr, ..., or cddddr of pair
libraries: (rnrs base), (rnrs)

These procedures are defined as the composition of up to four cars and cdrs. The a's and d's between the c and r represent the application of car or cdr in order from right to left. For example, the procedure cadr applied to a pair yields the car of the cdr of the pair and is equivalent to

(lambda (x) (car (cdr x))).

(caar '((a))) ⇒ a
(cadr '(a b c)) ⇒ b
(cdddr '(a b c d)) ⇒ (d)
(cadadr '((a b c))) ⇒ c

procedure: (list obj ...)
returns: a list of obj ...
libraries: (rnrs base), (rnrs)

list is equivalent to (lambda x x).

(list) ⇒ ()
(list 1 2 3) ⇒ (1 2 3)
(list 3 2 1) ⇒ (3 2 1)

procedure: (cons* obj ... final-obj)
returns: a list of obj ... terminated by final-obj
libraries: (rnrs lists), (rnrs)

If the objects obj ... are omitted, the result is simply final-obj. Otherwise, a list of obj ... is constructed, as with list, except that the final cdr field is final-obj instead of (). If final-obj is not a list, the result is an improper list.

(cons* '() ⇒ ()
(cons* '(a b)) ⇒ (a b)
procedure: \((\text{list? } \text{obj})\)
returns: \(\#\text{t}\) if \(\text{obj}\) is a proper list, \(\#\text{f}\) otherwise
libraries: \((\text{rnrs base}), (\text{rnrs})\)

\(\text{list?}\) must return \(\#\text{f}\) for all improper lists, including cyclic lists. A definition of \(\text{list?}\) is shown on page 67.

\[(\text{list? '()}) \Rightarrow \#\text{t}\]
\[(\text{list? '(a b c)}) \Rightarrow \#\text{t}\]
\[(\text{list? 'a}) \Rightarrow \#\text{f}\]
\[(\text{list? '(3 . 4)}) \Rightarrow \#\text{f}\]
\[(\text{list? 3}) \Rightarrow \#\text{f}\]
\[(\text{let ([x (list 'a 'b 'c)])}
  (\text{set-cdr! (cddr x) x})
  (\text{list? x})) \Rightarrow \#\text{f}\]

procedure: \((\text{length } \text{list})\)
returns: the number of elements in \(\text{list}\)
libraries: \((\text{rnrs base}), (\text{rnrs})\)

\(\text{length}\) may be defined as follows, using an adaptation of the hare and tortoise algorithm used for the definition of \(\text{list?}\) on page 67.

\[(\text{define length}
  (\text{lambda (x)}
    (\text{define improper-list}
      (\text{lambda ()}
        (\text{assertion-violation 'length "not a proper list" x})))
    (let f ([h x] [t x] [n 0])
      (if (pair? h)
        (let ([h (cdr h)])
          (if (pair? h)
            (if (eq? h t)
              (improper-list)
              (f (cdr h) (cdr t) (+ n 2)))
            (if (null? h)
              (+ n 1)
              (improper-list))))
        (if (null? h)
          n
          (improper-list)))))
  \))

\[(\text{length '()}) \Rightarrow 0\]
\[(\text{length '(a b c)}) \Rightarrow 3\]
\[(\text{length '(a b . c)}) \Rightarrow \text{exception}\]
\[(\text{length}
  (\text{let ([ls (list 'a 'b)])}
    (\text{set-cdr! (cddr ls) ls} \Rightarrow \text{exception}
      ls))
  )\]
(let ([ls (list 'a 'b)])
  (set-car! (cdr ls) ls) ⇒ 2
  ls))

**procedure**: (list-ref list n)
**returns**: the $n$th element (zero-based) of list
**libraries**: (rnrs base),(rnrs)

$n$ must be an exact nonnegative integer less than the length of list. list-ref may be defined without error checks as follows.

(define list-ref
  (lambda (ls n)
    (if (= n 0)
      (car ls)
      (list-ref (cdr ls) (- n 1))))))

(list-ref '(a b c) 0) ⇒ a
(list-ref '(a b c) 1) ⇒ b
(list-ref '(a b c) 2) ⇒ c

**procedure**: (list-tail list n)
**returns**: the $n$th tail (zero-based) of list
**libraries**: (rnrs base),(rnrs)

$n$ must be an exact nonnegative integer less than or equal to the length of list. The result is not a copy; the tail is eq? to the $n$th cdr of list (or to list itself, if $n$ is zero).

list-tail may be defined without error checks as follows.

(define list-tail
  (lambda (ls n)
    (if (= n 0)
      ls
      (list-tail (cdr ls) (- n 1))))))

(list-tail '(a b c) 0) ⇒ (a b c)
(list-tail '(a b c) 2) ⇒ (c)
(list-tail '(a b c) 3) ⇒ ()
(list-tail '(a b c . d) 2) ⇒ (c . d)
(list-tail '(a b c . d) 3) ⇒ d
(let ([x (list 1 2 3)])
  (eq? (list-tail x 2)
       (cddr x))) ⇒ #t

**procedure**: (append)
**procedure**: (append list ... obj)
**returns**: the concatenation of the input lists
**libraries**: (rnrs base),(rnrs)

append returns a new list consisting of the elements of the first list followed by the elements of the second list, the elements of the third list, and so on. The new list is made from new pairs for all arguments but the last; the last (which need not be a list) is merely placed at the end of the new structure. append may be defined without error checks as follows.
(define append
  (lambda args
    (let f ([ls '()] [args args])
      (if (null? args)
        ls
        (let g ([ls ls])
          (if (null? ls)
            (f (car args) (cdr args))
            (cons (car ls) (g (cdr ls)))))))))

(append '(a b c) '()) ⇒ (a b c)
(append '() '(a b c)) ⇒ (a b c)
(append '(a b) '(c d)) ⇒ (a b c d)
(append '(a b) 'c) ⇒ (a b . c)
(let ([x (list 'b)])
  (eq? x (cdr (append '(a) x)))) ⇒ #t

procedure: (reverse list)
returns: a new list containing the elements of list in reverse order
libraries: (rnrs base), (rnrs)

reverse may be defined without error checks as follows.

(define reverse
  (lambda (ls)
    (let rev ([ls ls] [new '()])
      (if (null? ls)
        new
        (rev (cdr ls) (cons (car ls) new))))))

(reverse '()) ⇒ ()
(reverse '(a b c)) ⇒ (c b a)

procedure: (memq obj list)
procedure: (memv obj list)
procedure: (member obj list)
returns: the first tail of list whose car is equivalent to obj, or #f
libraries: (rnrs lists), (rnrs)

These procedures traverse the argument list in order, comparing the elements of list against obj. If an object equivalent to obj is found, the tail of the list whose first element is that object is returned. If the list contains more than one object equivalent to obj, the first tail whose first element is equivalent to obj is returned. If no object equivalent to obj is found, #f is returned. The equivalence test for memq is eq?, for memv is eqv?, and for member is equal?:

(define memq
  (lambda (x ls)
    (cond
      [(null? ls) #f]
      [(eq? (car ls) x) ls]
      [else (memq x (cdr ls))])))
memv and member may be defined similarly, with eqv? and equal? in place of eq?.

(memq 'a '(b c a d e)) ⇒ (a d e)
(memq 'a '(b c d e g)) ⇒ #f
(memq 'a '(b a c a d a)) ⇒ (a c a d a)

(memv 3.4 '(1.2 2.3 3.4 4.5)) ⇒ (3.4 4.5)
(memv 3.4 '(1.3 2.5 3.7 4.9)) ⇒ #f
(let ([ls (list 'a 'b 'c)])
  (set-car! (memv 'b ls) 'z)
  ls) ⇒ (a z c)

(member '(b) '(((a) (b) (c))) ⇒ ((b) (c))
(member '(d) '(((a) (b) (c))) ⇒ #f
(member "b" '("a" "b" "c") ⇒ ("b" "c")

(let ()
  (define member?
    (lambda (x ls)
      (and (member x ls) #t)))
  (member? '(b) '(((a) (b) (c))) ⇒ #t

(define count-occurrences
  (lambda (x ls)
    (cond
      [(memq x ls) =>
        (lambda (ls)
          (+ (count-occurrences x (cdr ls)) 1))]
      [else 0])))

(count-occurrences 'a '(a b c d a)) ⇒ 2

**procedure:** (memp procedure list)
**returns:** the first tail of list for whose car procedure returns true, or #f
**libraries:** (rnrs lists), (rnrs)

**procedure** should accept one argument and return a single value. It should not modify list.

(memp odd? '(1 2 3 4)) ⇒ (1 2 3 4)
(memp even? '(1 2 3 4)) ⇒ (2 3 4)
(let ([ls (list 1 2 3 4)])
  (eq? (memp odd? ls) ls)) ⇒ #t
(let ([ls (list 1 2 3 4)])
  (eq? (memp even? ls) (cdr ls))) ⇒ #t
(memp odd? '(2 4 6 8)) ⇒ #f

**procedure:** (remq obj list)
**procedure:** (remv obj list)
**procedure:** (remove obj list)
**returns:** a list containing the elements of list with all occurrences of obj removed
**libraries:** (rnrs lists), (rnrs)

These procedures traverse the argument list, removing any objects that are equivalent to obj. The elements remaining in the output list are in the same order as they appear in the input list. If a tail of list
(including list itself) contains no occurrences of obj, the corresponding tail of the result list may be the same (by eq?) as the tail of the input list.

The equivalence test for remq is eq?, for remv is eqv?, and for remove is equal?.

(remp 'a '(a b a c a d)) ⇒ (b c)
(remp 'a '(b c d)) ⇒ (b c d)

(remv 1/2 '(1.2 1/2 0.5 3/2 4)) ⇒ (1.2 0.5 3/2 4)

(remove '(b) '(((a) (b) (c)))) ⇒ (a) (c))

procedure: (remp procedure list)
returns: a list of the elements of list for which procedure returns #f
libraries: (rnrs lists), (rnrs)

remp applies procedure to each element of list and returns a list containing only the elements for which procedure returns #f. The elements of the returned list appear in the same order as they appeared in the original list.

(remp odd? '(1 2 3 4)) ⇒ (2 4)
(remp
 (lambda (x) (and (> x 0) (< x 10)))
 '(-5 15 3 14 -20 6 0 -9)) ⇒ (-5 15 14 -20 0 -9)

procedure: (filter procedure list)
returns: a list of the elements of list for which procedure returns true
libraries: (rnrs lists), (rnrs)

filter applies procedure to each element of list and returns a new list containing only the elements for which procedure returns true. The elements of the returned list appear in the same order as they appeared in the original list.

(filter odd? '(1 2 3 4)) ⇒ (1 3)
(filter
 (lambda (x) (and (> x 0) (< x 10)))
 '(-5 15 3 14 -20 6 0 -9)) ⇒ (3 6)

procedure: (partition procedure list)
returns: see below
libraries: (rnrs lists), (rnrs)

partition applies procedure to each element of list and returns two values: a new list containing only the elements for which procedure returns true, and a new list containing only the elements for which procedure returns #f. The elements of the returned lists appear in the same order as they appeared in the original list.

(partition odd? '(1 2 3 4)) ⇒ (1 3)
(partition
  (lambda (x) (and (> x 0) (< x 10)))
  '(-5 15 3 14 -20 6 0 -9)) ⇒ (3 6)
  (-5 15 14 -20 0 -9)

The values returned by partition can be obtained by calling filter and remp separately, but this would require two calls to procedure for each element of list.

procedure: (find procedure list)
returns: the first element of list for which procedure returns true, or #f
libraries: (rnrs lists), (rnrs)

procedure should accept one argument and return a single value. It should not modify list.

find traverses the argument list in order, applying procedure to each element in turn. If procedure returns a true value for a given element, find returns that element without applying procedure to the remaining elements. If procedure returns #f for each element of list, find returns #f.

If a program must distinguish between finding #f in the list and finding no element at all, memp should be used instead.

(find odd? '(1 2 3 4)) ⇒ 1
(find even? '(1 2 3 4)) ⇒ 2
(find odd? '(2 4 6 8)) ⇒ #f
(find not '(1 a #f 55)) ⇒ #f

procedure: (assq obj alist)
procedure: (assv obj alist)
procedure: (assoc obj alist)
returns: first element of alist whose car is equivalent to obj, or #f
libraries: (rnrs lists), (rnrs)

The argument alist must be an association list. An association list is a proper list whose elements are key-value pairs of the form (key . value). Associations are useful for storing information (values) associated with certain objects (keys).

These procedures traverse the association list, testing each key for equivalence with obj. If an equivalent key is found, the key-value pair is returned. Otherwise, #f is returned.

The equivalence test for assq is eq?, for assv is eqv?, and for assoc is equal?. assq may be defined without error checks as follows.

(define assq
  (lambda (x ls)
    (cond
      [(null? ls) #f]
      [(eq? (caar ls) x) (car ls)]
      [else (assq x (cdr ls))])))

assv and assoc may be defined similarly, with eqv? and equal? in place of eq?.

(assq 'b '((a . 1) (b . 2))) ⇒ (b . 2)
(cdr (assq 'b '((a . 1) (b . 2)))) ⇒ 2

The Scheme Programming Language, 4th Edition
(2 4)

Section 6.3. Lists and Pairs
The interpreter given in Section 12.7 represents environments as association lists and uses \texttt{assq} for both variable lookup and assignment.

\textbf{procedure}: \texttt{(assq procedure alist)}

\textbf{returns}: first element of \texttt{alist} for whose car \texttt{procedure} returns true, or \#f

\textbf{libraries}: (rnrs lists), (rnrs)

\texttt{alist} must be an \textit{association list}. An association list is a proper list whose elements are key-value pairs of the form (key . value). \texttt{procedure} should accept one argument and return a single value. It should not modify \texttt{list}.

\begin{verbatim}
(assq odd? '((1 . a) (2 . b))) ⇒ (1 . a)
(assq even? '((1 . a) (2 . b))) ⇒ (2 . b)
(let ([ls (list (cons 1 'a) (cons 2 'b))])
   (eq? (assq odd? ls) (car ls))) ⇒ #t
(let ([ls (list (cons 1 'a) (cons 2 'b))])
   (eq? (assq even? ls) (cadr ls))) ⇒ #t
(assq odd? '((2 . b))) ⇒ #f
\end{verbatim}

\textbf{procedure}: \texttt{(list-sort predicate list)}

\textbf{returns}: a list containing the elements of \texttt{list} sorted according to \texttt{predicate}

\textbf{libraries}: (rnrs sorting), (rnrs)

\texttt{predicate} should be a procedure that expects two arguments and returns \#t if its first argument must precede its second in the sorted list. That is, if \texttt{predicate} is applied to two elements \texttt{x} and \texttt{y}, where \texttt{x} appears after \texttt{y} in the input list, it should return true only if \texttt{x} should appear before \texttt{y} in the output list. If this constraint is met, \texttt{list-sort} performs a stable sort, i.e., two elements are reordered only when necessary according to \texttt{predicate}. Duplicate elements are not removed. This procedure may call \texttt{predicate} up to $n \log n$ times, where \texttt{n} is the length of \texttt{list}.

\begin{verbatim}
(list-sort < '(3 4 2 1 2 5)) ⇒ (1 2 2 3 4 5)
(list-sort > '(0.5 1/2)) ⇒ (0.5 1/2)
(list-sort > '(1/2 0.5)) ⇒ (1/2 0.5)
(list->string
   (list-sort char?>
      (string->list "hello"))) ⇒ "ollhe"
\end{verbatim}

\textbf{Section 6.4. Numbers}

Scheme numbers may be classified as integers, rational numbers, real numbers, or complex numbers. This classification is hierarchical, in that all integers are rational, all rational numbers are real, and all real numbers are complex. The predicates \texttt{integer?}, \texttt{rational?}, \texttt{real?}, and \texttt{complex?} described in Section 6.2 are
used to determine into which of these classes a number falls.

A Scheme number may also be classified as exact or inexact, depending upon the quality of operations used to derive the number and the inputs to these operations. The predicates exact? and inexact? may be used to determine the exactness of a number. Most operations on numbers in Scheme are exactness preserving: if given exact operands they return exact values, and if given inexact operands or a combination of exact and inexact operands they return inexact values.

Exact integer and rational arithmetic is typically supported to arbitrary precision; the size of an integer or of the denominator or numerator of a ratio is limited only by system storage constraints. Although other representations are possible, inexact numbers are typically represented by floating-point numbers supported by the host computer's hardware or by system software. Complex numbers are typically represented as ordered pairs (real-part, imag-part), where real-part and imag-part are exact integers, exact rationals, or floating-point numbers.

Scheme numbers are written in a straightforward manner not much different from ordinary conventions for writing numbers. An exact integer is normally written as a sequence of numerals preceded by an optional sign. For example, 3, +19, −100000, and 208423089237489374 all represent exact integers.

An exact rational number is normally written as two sequences of numerals separated by a slash (/) and preceded by an optional sign. For example, 3/4, −6/5, and 1/1208203823 are all exact rational numbers. A ratio is reduced immediately to lowest terms when it is read and may in fact reduce to an exact integer.

Inexact real numbers are normally written in either floating-point or scientific notation. Floating-point notation consists of a sequence of numerals followed by a decimal point and another sequence of numerals, all preceded by an optional sign. Scientific notation consists of an optional sign, a sequence of numerals, an optional decimal point followed by a second string of numerals, and an exponent; an exponent is written as the letter e followed by an optional sign and a sequence of numerals. For example, 1.0 and −200.0 are valid inexact integers, and 1.5, 0.034, −10e−10 and 1.5e−5 are valid inexact rational numbers. The exponent is the power of ten by which the number preceding the exponent should be scaled, so that 2e3 is equivalent to 2000.0.

A mantissa width \(|w|\) may appear as the suffix of a real number or the real components of a complex number written in floating-point or scientific notation. The mantissa width \(m\) represents the number of significant bits in the representation of the number. The mantissa width defaults to 53, the number of significant bits in a normalized IEEE double floating-point number, or more. For denormalized IEEE double floating-point numbers, the mantissa width is less than 53. If an implementation cannot represent a number with the mantissa width specified, it uses a representation with at least as many significant bits as requested if possible, otherwise it uses its representation with the largest mantissa width.

Exact and inexact real numbers are written as exact or inexact integers or rational numbers; no provision is made in the syntax of Scheme numbers for nonrational real numbers, i.e., irrational numbers.

Complex numbers may be written in either rectangular or polar form. In rectangular form, a complex number is written as \(x+yi\) or \(x−yi\), where \(x\) is an integer, rational, or real number and \(y\) is an unsigned integer, rational, or real number. The real part, \(x\), may be omitted, in which case it is assumed to be zero. For example, \(3+4i, 3.2−3/4i, +i, \) and \(-3e-5i\) are complex numbers written in rectangular form. In polar form, a complex number is written as \(x@y\), where \(x\) and \(y\) are integer, rational, or real numbers. For example, \(1.101.764\) and \(-10−1/2\) are complex numbers written in polar form.

The syntaxes +inf.0 and -inf.0 represent inexact real numbers that represent positive and negative infinity. The syntaxes +nan.0 and -nan.0 represent an inexact "not-a-number" (NaN) value. Infinities may be produced by dividing inexact positive and negative values by inexact zero, and NaNs may also be produced by dividing inexact zero by inexact zero, among other ways.
The exactness of a numeric representation may be overridden by preceding the representation by either #e or #i. #e forces the number to be exact, and #i forces it to be inexact. For example, 1, #e1, 1/1, #e1/1, #e1.0, and #e1e0 all represent the exact integer 1, and #i3/10, 0.3, #i0.3, and 3e-1 all represent the inexact rational 0.3.

Numbers are written by default in base 10, although the special prefixes #b (binary), #o (octal), #d (decimal), and #x (hexadecimal) can be used to specify base 2, base 8, base 10, or base 16. For radix 16, the letters a through f or A through F serve as the additional numerals required to express digit values 10 through 15. For example, #b10101 is the binary equivalent of 2110, #o72 is the octal equivalent of 5810, and #x7c7 is the hexadecimal equivalent of 19910. Numbers written in floating-point and scientific notations are always written in base 10.

If both are present, radix and exactness prefixes may appear in either order.

A Scheme implementation may support more than one size of internal representation for inexact quantities. The exponent markers s (short), f (single), d (double), and l (long) may appear in place of the default exponent marker e to override the default size for numbers written in scientific notation. In implementations that support multiple representations, the default size has at least as much precision as double.

A precise grammar for Scheme numbers is given on page 459.

Any number can be written in a variety of different ways, but the system printer (invoked by put-datum, write, and display) and number->string express numbers in a compact form, using the fewest number of digits necessary to retain the property that, when read, the printed number is identical to the original number.

The remainder of this section describes "generic arithmetic" procedures that operate on numbers. The two sections that follow this section describe operations specific to fixnums and flonums, which are representations of exact, fixed-precision integer values and inexact real values.

The types of numeric arguments accepted by the procedures in this section are implied by the names given to the arguments: num for complex numbers (that is, all numbers), real for real numbers, rat for rational numbers, and int for integers. If a real, rat, or int is required, the argument must be considered real, rational, or integral by real?, rational?, or integer?, i.e., the imaginary part of the number must be exactly zero. Where exact integers are required, the name exint is used. In each case, a suffix may appear on the name, e.g., int2.

procedure: (exact? num)
returns: #t if num is exact, #f otherwise
libraries: (rnrs base), (rnrs)

(exact? 1) ⇒ #t
(exact? -15/16) ⇒ #t
(exact? 2.01) ⇒ #f
(exact? #i177) ⇒ #f
(exact? #i2/3) ⇒ #f
(exact? 1.0-2i) ⇒ #f

procedure: (inexact? num)
returns: #t if num is inexact, #f otherwise
libraries: (rnrs base), (rnrs)

(inexact? -123) ⇒ #f
(inexact? #i1123) ⇒ #t
The Scheme Programming Language, 4th Edition

(inexact? 1e23) ⇒ #t
(inexact? +i) ⇒ #f

procedure: (= num1 num2 num3 ...)
procedure: (< real1 real2 real3 ...)
procedure: (> real1 real2 real3 ...)
procedure: (<= real1 real2 real3 ...)
procedure: (>= real1 real2 real3 ...)
returns: #t if the relation holds, #f otherwise
libraries: (rnrs base), (rnrs)

The predicate = returns #t if its arguments are equal. The predicate < returns #t if its arguments are monotonically increasing, i.e., each argument is greater than the preceding ones, while > returns #t if its arguments are monotonically decreasing. The predicate <= returns #t if its arguments are monotonically nondecreasing, i.e., each argument is not less than the preceding ones, while >= returns #t if its arguments are monotonically nonincreasing.

As implied by the names of the arguments, = is defined for complex arguments while the other relational predicates are defined only for real arguments. Two complex numbers are considered equal if their real and imaginary parts are equal. Comparisons involving NaNs always return #f.

(= 7 7) ⇒ #t
(= 7 9) ⇒ #f
(< 2e3 3e2) ⇒ #f
(<= 1 2 3 3 4 5) ⇒ #t
(<= 1 2 3 4 5) ⇒ #t

(> 1 2 2 3 3 4) ⇒ #f
(>= 1 2 2 3 3 4) ⇒ #f

(= -1/2 -0.5) ⇒ #t
(= 2/3 .667) ⇒ #f
(= 7.2+0i 7.2) ⇒ #t
(= 7.2-3i 7) ⇒ #f

(< 1/2 2/3 3/4) ⇒ #t
(> 8 4.102 2/3 -5) ⇒ #f

(let ([x 0.218723452])
  (< 0.210 x 0.220)) ⇒ #t

(let ([i 1] [v (vector 'a 'b 'c)])
  (< -1 i (vector-length v))) ⇒ #t

(apply < '(1 2 3 4)) ⇒ #t
(apply > '(4 3 3 2)) ⇒ #f

(= +nan.0 +nan.0) ⇒ #f
(< +nan.0 +nan.0) ⇒ #f
(> +nan.0 +nan.0) ⇒ #f
(>= +inf.0 +nan.0) ⇒ #f
(> +nan.0 -inf.0) ⇒ #f
(> +nan.0 0.0) ⇒ #f
procedure: (+ num ...)
returns: the sum of the arguments num ...
libraries: (rnrs base), (rnrs)

When called with no arguments, + returns 0.

(+) ⇒ 0
(+ 1 2) ⇒ 3
(+ 1/2 2/3) ⇒ 7/6
(+ 3 4 5) ⇒ 12
(+ 3.0 4) ⇒ 7.0
(+ 3+4i 4+3i) ⇒ 7+7i
(apply + '(1 2 3 4 5)) ⇒ 15

procedure: (- num)
returns: the additive inverse of num
procedure: (- num1 num2 num3 ...)
returns: the difference between num1 and the sum of num2 num3 ...
libraries: (rnrs base), (rnrs)

(- 3) ⇒ -3
(- -2/3) ⇒ 2/3
(- 4 3.0) ⇒ 1.0
(- 3.25+4.25i 1/4+1/4i) ⇒ 3.0+4.0i
(- 4 3 2 1) ⇒ -2

procedure: (* num ...)
returns: the product of the arguments num ...
libraries: (rnrs base), (rnrs)

(*) ⇒ 1
(* 3.4) ⇒ 3.4
(* 1 1/2) ⇒ 1/2
(* 3 4 5.5) ⇒ 66.0
(* 1+2i 3+4i) ⇒ -5+10i
(apply * '(1 2 3 4 5)) ⇒ 120

procedure: (/ num)
returns: the multiplicative inverse of num
procedure: (/ num1 num2 num3 ...)
returns: the result of dividing num1 by the product of num2 num3 ...
libraries: (rnrs base), (rnrs)

(/ -17) ⇒ -1/17
(/ 1/2) ⇒ 2
(/ .5) ⇒ 2.0
(/ 3 4) ⇒ 3/4
(/ 3.0 4) ⇒ .75
(/ -5+10i 3+4i) ⇒ 1+2i
(/ 60 5 4 3 2) ⇒ 1/2

procedure: (zero? num)
returns: #t if num is zero, #f otherwise
libraries: (rnrs base), (rnrs)

zero? is equivalent to (lambda (x) (= x 0)).

(zero? 0) ⇒ #t
(zero? 1) ⇒ #f
(zero? (- 3.0 3.0)) ⇒ #t
(zero? (+ 1/2 1/2)) ⇒ #f
(zero? 0+0i) ⇒ #t
(zero? 0.0-0.0i) ⇒ #t

procedure: (positive? real)
returns: #t if real is greater than zero, #f otherwise
libraries: (rnrs base), (rnrs)

positive? is equivalent to (lambda (x) (> x 0)).

(positive? 128) ⇒ #t
(positive? 0.0) ⇒ #f
(positive? 1.8e-15) ⇒ #t
(positive? -2/3) ⇒ #f
(positive? .001-0.0i) ⇒ exception: not a real number

procedure: (negative? real)
returns: #t if real is less than zero, #f otherwise
libraries: (rnrs base), (rnrs)

negative? is equivalent to (lambda (x) (< x 0)).

(negative? -65) ⇒ #t
(negative? 0) ⇒ #f
(negative? -0.0121) ⇒ #t
(negative? 15/16) ⇒ #f
(negative? -7.0+0.0i) ⇒ exception: not a real number

procedure: (even? int)
returns: #t if int is even, #f otherwise

procedure: (odd? int)
returns: #t if int is odd, #f otherwise
libraries: (rnrs base), (rnrs)

(even? 0) ⇒ #t
(even? 1) ⇒ #f
(even? 2.0) ⇒ #t
(even? -120762398465) ⇒ #f
(even? 2.0+0.0i) ⇒ exception: not an integer

(odd? 0) ⇒ #f
(odd? 1) ⇒ #t
(odd? 2.0) ⇒ #f
(odd? -120762398465) ⇒ #t
(odd? 2.0+0.0i) ⇒ exception: not an integer
procedure: (finite? real)
returns: #t if real is finite, #f otherwise
procedure: (infinite? real)
returns: #t if real is infinite, #f otherwise
procedure: (nan? real)
returns: #t if real is a NaN, #f otherwise
libraries: (rnrs base), (rnrs)

(finite? 2/3) ⇒ #t
(infinite? 2/3) ⇒ #f
(nan? 2/3) ⇒ #f

(finite? 3.1415) ⇒ #t
(infinite? 3.1415) ⇒ #f
(nan? 3.1415) ⇒ #f

(finite? +inf.0) ⇒ #f
(infinite? -inf.0) ⇒ #t
(nan? -inf.0) ⇒ #f

(finite? +nan.0) ⇒ #f
(infinite? +nan.0) ⇒ #f
(nan? +nan.0) ⇒ #t

procedure: (quotient int1 int2)
returns: the integer quotient of int1 and int2
procedure: (remainder int1 int2)
returns: the integer remainder of int1 and int2
procedure: (modulo int1 int2)
returns: the integer modulus of int1 and int2
libraries: (rnrs r5rs)

The result of remainder has the same sign as int1, while the result of modulo has the same sign as int2.

(quotient 45 6) ⇒ 7
(quotient 6.0 2.0) ⇒ 3.0
(quotient 3.0 -2) ⇒ -1.0

(remainder 16 4) ⇒ 0
(remainder 5 2) ⇒ 1
(remainder -45.0 7) ⇒ -3.0
(remainder 10.0 -3.0) ⇒ 1.0
(remainder -17 -9) ⇒ -8

(modulo 16 4) ⇒ 0
(modulo 5 2) ⇒ 1
(modulo -45.0 7) ⇒ 4.0
(modulo 10.0 -3.0) ⇒ -2.0
(modulo -17 -9) ⇒ -8

procedure: (div x1 x2)
procedure: (mod x1 x2)
procedure: (div-and-mod x1 x2)
returns: see below
If \( x_1 \) and \( x_2 \) are exact, \( x_2 \) must not be zero. These procedures implement number-theoretic integer division, with the \( \text{div} \) operation being related to quotient and the \( \text{mod} \) operation being related to remainder or modulo, but in both cases extended to handle real numbers.

The value \( n_d \) of \( \text{div} \ x_1 \ x_2 \) is an integer, and the value \( x_m \) of \( \text{mod} \ x_1 \ x_2 \) is a real number such that \( x_1 = n_d \cdot x_2 + x_m \) and \( 0 \leq x_m < |x_2| \). In situations where the implementation cannot represent the mathematical results prescribed by these equations as a number object, \( \text{div} \) and \( \text{mod} \) return an unspecified number or raise an exception with condition type \&implementation-restriction.

The \( \text{div-and-mod} \) procedure behaves as if defined as follows.

\[
\text{(define (div-and-mod x1 x2) (values (div x1 x2) (mod x1 x2)))}
\]

That is, unless it raises an exception in the circumstance described above, it returns two values: the result of calling \( \text{div} \) on the two arguments and the result of calling \( \text{mod} \) on the two arguments.

\[
\begin{align*}
(\text{div} 17 3) & \Rightarrow 5 \\
(\text{mod} 17 3) & \Rightarrow 2 \\
(\text{div} -17 3) & \Rightarrow -6 \\
(\text{mod} -17 3) & \Rightarrow 1 \\
(\text{div} 17 -3) & \Rightarrow -5 \\
(\text{mod} 17 -3) & \Rightarrow 2 \\
(\text{div} -17 -3) & \Rightarrow 6 \\
(\text{mod} -17 -3) & \Rightarrow 1 \\
\end{align*}
\]

\[
\begin{align*}
(\text{div-and-mod} 17.5 3) & \Rightarrow 5.0 \\
& \quad 2.5
\end{align*}
\]

\textbf{procedure:} \( \text{div0} \ x_1 \ x_2 \)

\textbf{procedure:} \( \text{mod0} \ x_1 \ x_2 \)

\textbf{procedure:} \( \text{div0-and-mod0} \ x_1 \ x_2 \)

\textbf{returns:} see below

\textbf{libraries:} (rnrs base), (rnrs)

If \( x_1 \) and \( x_2 \) are exact, \( x_2 \) must not be zero. These procedures are similar to \( \text{div}, \text{mod}, \) and \( \text{div-and-mod}, \) but constrain the "mod" value differently, which also affects the "div" value. The value \( n_d \) of \( \text{div0} \ x_1 \ x_2 \) is an integer, and the value \( x_m \) of \( \text{mod0} \ x_1 \ x_2 \) is a real number such that \( x_1 = n_d \cdot x_2 + x_m \) and \(-|x_2|/2| \leq x_m < |x_2|/2|\). In situations where the implementation cannot represent the mathematical results prescribed by these equations as a number object, \( \text{div0} \) and \( \text{mod0} \) return an unspecified number or raise an exception with condition type \&implementation-restriction.

The \( \text{div0-and-mod0} \) procedure behaves as if defined as follows.

\[
\text{(define (div0-and-mod0 x1 x2) (values (div0 x1 x2) (mod0 x1 x2)))}
\]

That is, unless it raises an exception in the circumstance described above, it returns two values: the result of calling \( \text{div0} \) on the two arguments and the result of calling \( \text{mod0} \) on the two arguments.

\[
\begin{align*}
(\text{div0} 17 3) & \Rightarrow 6 \\
(\text{mod0} 17 3) & \Rightarrow -1 \\
(\text{div0} -17 3) & \Rightarrow -6 \\
(\text{mod0} -17 3) & \Rightarrow 1 \\
\end{align*}
\]
\[(\text{div0} 17 -3) \Rightarrow -6\]
\[(\text{mod0} 17 -3) \Rightarrow -1\]
\[(\text{div0} -17 -3) \Rightarrow 6\]
\[(\text{mod0} -17 -3) \Rightarrow 1\]

\[(\text{div0-and-mod0} 17.5 3) \Rightarrow 6.0\]
\[-0.5\]

**procedure:** (truncate real)
**returns:** the integer closest to real toward zero
**libraries:** (rnrs base), (rnrs)

If real is an infinity or NaN, truncate returns real.

\[(\text{truncate} 19) \Rightarrow 19\]
\[(\text{truncate} 2/3) \Rightarrow 0\]
\[(\text{truncate} -2/3) \Rightarrow 0\]
\[(\text{truncate} 17.3) \Rightarrow 17.0\]
\[(\text{truncate} -17/2) \Rightarrow -8\]

**procedure:** (floor real)
**returns:** the integer closest to real toward \(-\infty\)
**libraries:** (rnrs base), (rnrs)

If real is an infinity or NaN, floor returns real.

\[(\text{floor} 19) \Rightarrow 19\]
\[(\text{floor} 2/3) \Rightarrow 0\]
\[(\text{floor} -2/3) \Rightarrow -1\]
\[(\text{floor} 17.3) \Rightarrow 17.0\]
\[(\text{floor} -17/2) \Rightarrow -9\]

**procedure:** (ceiling real)
**returns:** the integer closest to real toward \(+\infty\)
**libraries:** (rnrs base), (rnrs)

If real is an infinity or NaN, ceiling returns real.

\[(\text{ceiling} 19) \Rightarrow 19\]
\[(\text{ceiling} 2/3) \Rightarrow 1\]
\[(\text{ceiling} -2/3) \Rightarrow 0\]
\[(\text{ceiling} 17.3) \Rightarrow 18.0\]
\[(\text{ceiling} -17/2) \Rightarrow -8\]

**procedure:** (round real)
**returns:** the integer closest to real
**libraries:** (rnrs base), (rnrs)

If real is exactly between two integers, the closest even integer is returned. If real is an infinity or NaN, round returns real.

\[(\text{round} 19) \Rightarrow 19\]
\[(\text{round} 2/3) \Rightarrow 1\]
\[(\text{round} -2/3) \Rightarrow -1\]
(round 17.3) ⇒ 17.0  
(round -17/2) ⇒ -8  
(round 2.5) ⇒ 2.0  
(round 3.5) ⇒ 4.0  

**procedure**: (abs real)  
**returns**: the absolute value of real  
**libraries**: (rnrs base), (rnrs)  

abs is equivalent to (lambda (x) (if (< x 0) (- x) x)). abs and magnitude (see page 183) are identical for real inputs.

(abs 1) ⇒ 1  
(abs -3/4) ⇒ 3/4  
(abs 1.83) ⇒ 1.83  
(abs -0.093) ⇒ 0.093  

**procedure**: (max real₁ real₂ ...)  
**returns**: the maximum of real₁ real₂ ...  
**libraries**: (rnrs base), (rnrs)  

(max 4 -7 2 0 -6) ⇒ 4  
(max 1/2 3/4 4/5 5/6 6/7) ⇒ 6/7  
(max 1.5 1.3 -0.3 0.4 2.0 1.8) ⇒ 2.0  
(max 5 2.0) ⇒ 5.0  
(max -5 -2.0) ⇒ -2.0  
(let ([ls '(7 3 5 2 9 8)])  
  (apply max ls)) ⇒ 9  

**procedure**: (min real₁ real₂ ...)  
**returns**: the minimum of real₁ real₂ ...  
**libraries**: (rnrs base), (rnrs)  

(min 4 -7 2 0 -6) ⇒ -7  
(min 1/2 3/4 4/5 5/6 6/7) ⇒ 1/2  
(min 1.5 1.3 -0.3 0.4 2.0 1.8) ⇒ -0.3  
(min 5 2.0) ⇒ 2.0  
(min -5 -2.0) ⇒ -5.0  
(let ([ls '(7 3 5 2 9 8)])  
  (apply min ls)) ⇒ 2  

**procedure**: (gcd int ...)  
**returns**: the greatest common divisor of its arguments int ...  
**libraries**: (rnrs base), (rnrs)  

The result is always nonnegative, i.e., factors of -1 are ignored. When called with no arguments, gcd returns 0.

(gcd) ⇒ 0  
(gcd 34) ⇒ 34  
(gcd 33.0 15.0) ⇒ 3.0  
(gcd 70 -42 28) ⇒ 14  

**procedure**: (lcm int ...)
returns: the least common multiple of its arguments \( \text{int...} \)

libraries: \((\text{rnrs base}), (\text{rnrs})\)

The result is always nonnegative, i.e., common multiples of -1 are ignored. Although \(\text{lcm}\) should probably return \(\infty\) when called with no arguments, it is defined to return 1. If one or more of the arguments is 0, \(\text{lcm}\) returns 0.

\[
\begin{align*}
(\text{lcm}) & \Rightarrow 1 \\
(\text{lcm 34}) & \Rightarrow 34 \\
(\text{lcm 33.0 15.0}) & \Rightarrow 165.0 \\
(\text{lcm 70 -42 28}) & \Rightarrow 420 \\
(\text{lcm 17.0 0}) & \Rightarrow 0.0
\end{align*}
\]

procedure: \((\text{expt num}_1, \text{num}_2)\)
returns: \(\text{num}_1\) raised to the \(\text{num}_2\) power

libraries: \((\text{rnrs base}), (\text{rnrs})\)

If both arguments are 0, \(\text{expt}\) returns 1.

\[
\begin{align*}
(\text{expt 2 10}) & \Rightarrow 1024 \\
(\text{expt 2 -10}) & \Rightarrow 1/1024 \\
(\text{expt 2 -10.0}) & \Rightarrow 9.765625e-4 \\
(\text{expt -1/2 5}) & \Rightarrow -1/32 \\
(\text{expt 3.0 3}) & \Rightarrow 27.0 \\
(\text{expt +i 2}) & \Rightarrow -1
\end{align*}
\]

procedure: \((\text{inexact num})\)
returns: an inexact representation of \(\text{num}\)

libraries: \((\text{rnrs base}), (\text{rnrs})\)

If \(\text{num}\) is already inexact, it is returned unchanged. If no inexact representation for \(\text{num}\) is supported by the implementation, an exception with condition type \&implementation-violation\ may be raised. \(\text{inexact}\) may also return \(+inf.0\) or \(-inf.0\) for inputs whose magnitude exceeds the range of the implementation's inexact number representations.

\[
\begin{align*}
(\text{inexact 3}) & \Rightarrow 3.0 \\
(\text{inexact 3.0}) & \Rightarrow 3.0 \\
(\text{inexact -1/4}) & \Rightarrow -.25 \\
(\text{inexact 3+4i}) & \Rightarrow 3.0+4.0i \\
(\text{inexact (expt 10 20)}) & \Rightarrow 1e20
\end{align*}
\]

procedure: \((\text{exact num})\)
returns: an exact representation of \(\text{num}\)

libraries: \((\text{rnrs base}), (\text{rnrs})\)

If \(\text{num}\) is already exact, it is returned unchanged. If no exact representation for \(\text{num}\) is supported by the implementation, an exception with condition type \&implementation-violation\ may be raised.

\[
\begin{align*}
(\text{exact 3.0}) & \Rightarrow 3 \\
(\text{exact 3}) & \Rightarrow 3 \\
(\text{exact -.25}) & \Rightarrow -1/4 \\
(\text{exact 3.0+4.0i}) & \Rightarrow 3+4i \\
(\text{exact 1e20}) & \Rightarrow 10000000000000000000
\end{align*}
\]
procedure: (exact->inexact num)
returns: an inexact representation of num

procedure: (inexact->exact num)
returns: an exact representation of num

libraries: (rnrs r5rs)

These are alternative names for inexact and exact, supported for compatibility with the Revised 5 Report.

procedure: (rationalize real₁ real₂)
returns: see below
libraries: (rnrs base), (rnrs)

rationalize returns the simplest rational number that differs from real₁ by no more than real₂. A rational number \( q_1 = \frac{n_1}{m_1} \) is simpler than another rational number \( q_2 = \frac{n_2}{m_2} \) if \( |n_1| \leq |n_2| \) and \( |m_1| \leq |m_2| \) and either \( |n_1| < |n_2| \) or \( |m_1| < |m_2| \).

(rationalize 3/10 1/10) ⇒ 1/3
(rationalize .3 1/10) ⇒ 0.3333333333333333
(eqv? (rationalize .3 1/10) #i1/3) ⇒ #t

procedure: (numerator rat)
returns: the numerator of rat
libraries: (rnrs base), (rnrs)

If rat is an integer, the numerator is rat.

(numerator 9) ⇒ 9
(numerator 9.0) ⇒ 9.0
(numerator 0.0) ⇒ 0.0
(numerator 2/3) ⇒ 2
(numerator -9/4) ⇒ -9
(numerator -2.25) ⇒ -9.0

procedure: (denominator rat)
returns: the denominator of rat
libraries: (rnrs base), (rnrs)

If rat is an integer, including zero, the denominator is one.

(denominator 9) ⇒ 1
(denominator 9.0) ⇒ 1.0
(denominator 0) ⇒ 1
(denominator 0.0) ⇒ 1.0
(denominator 2/3) ⇒ 3
(denominator -9/4) ⇒ 4
(denominator -2.25) ⇒ 4.0

procedure: (real-part num)
returns: the real component of num
libraries: (rnrs base), (rnrs)

If num is real, real-part returns num.

(real-part 3+4i) ⇒ 3
(real-part -2.3+0.7i) \Rightarrow -2.3 \\
(real-part -i) \Rightarrow 0 \\
(real-part 17.2) \Rightarrow 17.2 \\
(real-part -17/100) \Rightarrow -17/100 \\

procedure: (imag-part num) 
returns: the imaginary component of num 
libraries: (rnrs base), (rnrs) 

If num is real, imag-part returns exact zero. 

(imag-part 3+4i) \Rightarrow 4 \\
(imag-part -2.3+0.7i) \Rightarrow 0.7 \\
(imag-part -i) \Rightarrow -1 \\
(imag-part -2.5) \Rightarrow 0 \\
(imag-part -17/100) \Rightarrow 0 \\

procedure: (make-rectangular real\_1, real\_2) 
returns: a complex number with real component real\_1 and imaginary component real\_2 
libraries: (rnrs base), (rnrs) 

(make-rectangular -2 7) \Rightarrow -2+7i \\
(make-rectangular 2/3 -1/2) \Rightarrow 2/3-1/2i \\
(make-rectangular 3.2 5.3) \Rightarrow 3.2+5.3i \\

procedure: (make-polar real\_1, real\_2) 
returns: a complex number with magnitude real\_1 and angle real\_2 
libraries: (rnrs base), (rnrs) 

(make-polar 2 0) \Rightarrow 2 \\
(make-polar 2.0 0.0) \Rightarrow 2.0+0.0i \\
(make-polar 1.0 (asin -1.0)) \Rightarrow 0.0-1.0i \\
(eqv? (make-polar 7.2 -0.588) 7.2@-0.588) \Rightarrow \#t \\

procedure: (angle num) 
returns: the angle part of the polar representation of num 
libraries: (rnrs base), (rnrs) 

The range of the result is \( -\pi \) (exclusive) to \( +\pi \) (inclusive). 

(angle 7.3@1.5708) \Rightarrow 1.5708 \\
(angle 5.2) \Rightarrow 0.0 \\

procedure: (magnitude num) 
returns: the magnitude of num 
libraries: (rnrs base), (rnrs) 

magnitude and abs (see page 178) are identical for real arguments. The magnitude of a complex number \( x + yi \) is \( \sqrt{x^2 + y^2} \). 

(magnitude 1) \Rightarrow 1 \\
(magnitude 3/4) \Rightarrow 3/4 \\
(magnitude 1.83) \Rightarrow 1.83 \\
(magnitude -0.093) \Rightarrow 0.093
(magnitude 3+4i) ⇒ 5
(magnitude 7.25@1.5708) ⇒ 7.25

**procedure**: (sqrt num)

**returns**: the principal square root of num

**libraries**: (rnrs base), (rnrs)

Implementations are encouraged, but not required, to return exact results for exact inputs to sqrt whenever feasible.

(sqrt 16) ⇒ 4
(sqrt 1/4) ⇒ 1/2
(sqrt 4.84) ⇒ 2.2
(sqrt -4.84) ⇒ 0.0+2.2i
(sqrt 3+4i) ⇒ 2+1i
(sqrt -3.0-4.0i) ⇒ 1.0-2.0i

**procedure**: (exact-integer-sqrt n)

**returns**: see below

**libraries**: (rnrs base), (rnrs)

This procedure returns two nonnegative exact integers s and r where n = s^2 + r and n < (s + 1)^2.

(exact-integer-sqrt 0) ⇒ 0
  0
(exact-integer-sqrt 9) ⇒ 3
  ⇒ 0
(exact-integer-sqrt 19) ⇒ 4
  ⇒ 3

**procedure**: (exp num)

**returns**: e to the num power

**libraries**: (rnrs base), (rnrs)

(exp 0.0) ⇒ 1.0
(exp 1.0) ⇒ 2.7182818284590455
(exp -.5) ⇒ 0.6065306597126334

**procedure**: (log num)

**returns**: the natural logarithm of num

**procedure**: (log num1 num2)

**returns**: the base-num2 logarithm of num1

**libraries**: (rnrs base), (rnrs)

(log 1.0) ⇒ 0.0
(log (exp 1.0)) ⇒ 1.0
(/ (log 100) (log 10)) ⇒ 2.0
(log (make-polar (exp 2.0) 1.0)) ⇒ 2.0+1.0i

(log 100.0 10.0) ⇒ 2.0
(log .125 2.0) ⇒ -3.0

**procedure**: (sin num)

**procedure**: (cos num)
The argument is specified in radians.

\[
\begin{align*}
\sin 0.0 & \Rightarrow 0.0 \\
\cos 0.0 & \Rightarrow 1.0 \\
\tan 0.0 & \Rightarrow 0.0
\end{align*}
\]

\[
\begin{align*}
\sin 0.0 & \Rightarrow 0.0 \\
\cos 0.0 & \Rightarrow 1.0 \\
\tan 0.0 & \Rightarrow 0.0
\end{align*}
\]

\[
\begin{align*}
\sin 0.0 & \Rightarrow 0.0 \\
\cos 0.0 & \Rightarrow 1.0 \\
\tan 0.0 & \Rightarrow 0.0
\end{align*}
\]

The result is in radians. The arc sine and arc cosine of a complex number \( z \) are defined as follows.

\[
sin^{-1}(z) = -i \log(i z + \sqrt{1 - z^2})
\]

\[
cos^{-1}(z) = \pi/2 - sin^{-1}(z)
\]

\[
(\text{define pi (*/(asin 1) 2)) (*/(acos 0) 2) pi) \Rightarrow \#t
\]

When passed a single complex argument \( num \) (the first form), \( \text{atan} \) returns the arc tangent of \( num \). The arc tangent of a complex number \( z \) is defined as follows.

\[
\tan^{-1}(z) = \log(1 + iz) - \log(1 - iz) / (2i)
\]

When passed two real arguments (the second form), \( \text{atan} \) is equivalent to

\[
(\text{lambda} (y x) (\text{angle} (\text{make-rectangular} x y)))
\]

\[
(\text{define pi (*/(atan 1) 4)) (*/(atan 1.0 0.0) 2) pi) \Rightarrow \#t
\]

\[
(\text{define pi (*/(atan 1) 4)) (*/(atan 1.0 0.0) 2) pi) \Rightarrow \#t
\]

\[
(\text{define pi (*/(atan 1) 4)) (*/(atan 1.0 0.0) 2) pi) \Rightarrow \#t
\]

The inputs are treated as if represented in two's complement, even if they are not represented that way internally.

\[
(\text{bitwise-not 0}) \Rightarrow -1 \\
(\text{bitwise-not 3}) \Rightarrow -4
\]
(bitwise-and #b01101 #b00111) ⇒ #b00101
(bitwise-ior #b01101 #b00111) ⇒ #b01111
(bitwise-xor #b01101 #b00111) ⇒ #b01010

**procedure**: (bitwise-if exint₁ exint₂ exint₃)

**returns**: the bitwise "if" of its arguments

**libraries**: (rnrs arithmetic bitwise), (rnrs)

The inputs are treated as if represented in two's complement, even if they are not represented that way internally.

For each bit set in $exint_1$, the corresponding bit of the result is taken from $exint_2$, and for each bit not set in $exint_1$, the corresponding bit of the result is taken from $x_3$.

(bitwise-if #b101010 #b111000 #b001100) ⇒ #b101100

**bitwise-if** might be defined as follows:

```
(define bitwise-if
  (lambda (exint₁ exint₂ exint₃)
    (bitwise-ior
      (bitwise-and exint₁ exint₂)
      (bitwise-and (bitwise-not exint₁) exint₃))))
```

**procedure**: (bitwise-bit-count exint)

**returns**: see below

**libraries**: (rnrs arithmetic bitwise), (rnrs)

For nonnegative inputs, **bitwise-bit-count** returns the number of bits set in the two’s complement representation of $exint$. For negative inputs, it returns a negative number whose magnitude is one greater than the number of bits not set in the two’s complement representation of $exint$, which is equivalent to

(bitwise-not (bitwise-bit-count (bitwise-not exint))).

(bitwise-bit-count #b00000) ⇒ 0
(bitwise-bit-count #b00001) ⇒ 1
(bitwise-bit-count #b00100) ⇒ 1
(bitwise-bit-count #b10101) ⇒ 3

(bitwise-bit-count -1) ⇒ -1
(bitwise-bit-count -2) ⇒ -2
(bitwise-bit-count -4) ⇒ -3

**procedure**: (bitwise-length exint)

**returns**: see below

**libraries**: (rnrs arithmetic bitwise), (rnrs)

This procedure returns the number of bits of the smallest two’s complement representation of $exint$, not including the sign bit for negative numbers. For 0 **bitwise-length** returns 0.

(bitwise-length #b00000) ⇒ 0
(bitwise-length #b00001) ⇒ 1
(bitwise-length #b00100) ⇒ 3
(bitwise-length #b00110) ⇒ 3
procedure: (bitwise-first-bit-set exint)
returns: the index of the least significant bit set in exint
libraries: (rnrs arithmetic bitwise), (rnrs)

The input is treated as if represented in two's complement, even if it is not represented that way internally.

If exint is 0, bitwise-first-bit-set returns -1.

(bitwise-first-bit-set #b00000) ⇒ -1
(bitwise-first-bit-set #b00001) ⇒ 0
(bitwise-first-bit-set #b01100) ⇒ 2

(bitwise-first-bit-set -1) ⇒ 0
(bitwise-first-bit-set -2) ⇒ 1
(bitwise-first-bit-set -3) ⇒ 0

procedure: (bitwise-bit-set? exint_1 exint_2)
returns: #t if bit exint_2 of exint_1 is set, #f otherwise
libraries: (rnrs arithmetic bitwise), (rnrs)

exint_2 is taken as a zero-based index for the bits in the two's complement representation of exint_1. The two's complement representation of a nonnegative number conceptually extends to the left (toward more significant bits) with an infinite number of zero bits, and the two's complement representation of a negative number conceptually extends to the left with an infinite number of one bits. Thus, exact integers can be used to represent arbitrarily large sets, where 0 is the empty set, -1 is the universe, and bitwise-bit-set? is used to test for membership.

(bitwise-bit-set? #b01011 0) ⇒ #t
(bitwise-bit-set? #b01011 2) ⇒ #f

(bitwise-bit-set? -1 0) ⇒ #t
(bitwise-bit-set? -1 20) ⇒ #t
(bitwise-bit-set? -3 1) ⇒ #f

(bitwise-bit-set? 0 5000) ⇒ #f
(bitwise-bit-set? -1 5000) ⇒ #t

procedure: (bitwise-copy-bit exint_1 exint_2 exint_3)
returns: exint_1 with bit exint_2 replaced by exint_3
libraries: (rnrs arithmetic bitwise), (rnrs)

exint_2 is taken as a zero-based index for the bits in the two's complement representation of exint_1. exint_3 must be 0 or 1. This procedure effectively clears or sets the specified bit depending on the value of exint_3. exint_1 is treated as if represented in two's complement, even if it is not represented that way internally.

(bitwise-copy-bit #b01110 0 1) ⇒ #b01111
(bitwise-copy-bit #b01110 2 0) ⇒ #b01010
procedure: (bitwise-bit-field exint₁ exint₂ exint₃)
returns: see below
libraries: (rnrs arithmetic bitwise), (rnrs)

exint₁ and exint₃ must be nonnegative, and exint₂ must not be greater than exint₃. This procedure returns the number represented by extracting from exint₁, the sequence of bits from exint₂ (inclusive) to exint₃ (exclusive). exint₁ is treated as if represented in two's complement, even if it is not represented that way internally.

(bitwise-bit-field #b10110 0 3) ⇒ #b00110
(bitwise-bit-field #b10110 1 3) ⇒ #b00011
(bitwise-bit-field #b10110 2 3) ⇒ #b00001
(bitwise-bit-field #b10110 3 3) ⇒ #b00000

procedure: (bitwise-copy-bit-field exint₁ exint₂ exint₃ exint₄)
returns: see below
libraries: (rnrs arithmetic bitwise), (rnrs)

exint₂ and exint₃ must be nonnegative, and exint₄ must not be greater than exint₃. This procedure returns exint₁ with the n bits from exint₂ (inclusive) to exint₃ (exclusive) replaced by the low-order n bits of exint₄. exint₁ and exint₄ are treated as if represented in two's complement, even if they are not represented that way internally.

(bitwise-copy-bit-field #b10000 0 3 #b10101) ⇒ #b10101
(bitwise-copy-bit-field #b10000 1 3 #b10101) ⇒ #b10010
(bitwise-copy-bit-field #b10000 2 3 #b10101) ⇒ #b10100
(bitwise-copy-bit-field #b10000 3 3 #b10101) ⇒ #b10000

procedure: (bitwise-arithmetic-shift-right exint₁ exint₂)
returns: exint₁ arithmetically shifted right by exint₂ bits

procedure: (bitwise-arithmetic-shift-left exint₁ exint₂)
returns: exint₁ shifted left by exint₂ bits
libraries: (rnrs arithmetic bitwise), (rnrs)

exint₂ must be nonnegative. exint₁ is treated as if represented in two's complement, even if it is not represented that way internally.

(bitwise-arithmetic-shift-right #b10000 3) ⇒ #b00010
(bitwise-arithmetic-shift-right #b10000 -1 1) ⇒ -1
(bitwise-arithmetic-shift-right #b10000 -64 3) ⇒ -8

(bitwise-arithmetic-shift-left #b00010 2) ⇒ #b01000
(bitwise-arithmetic-shift-left #b00010 -1 2) ⇒ -4

procedure: (bitwise-arithmetic-shift exint₁ exint₂)
returns: see below
libraries: (rnrs arithmetic bitwise), (rnrs)

If exint₂ is negative, bitwise-arithmetic-shift returns the result of arithmetically shifting exint₁ right by exint₂ bits. Otherwise, bitwise-arithmetic-shift returns the result of shifting exint₁ left by exint₂ bits. exint₁ is treated as if represented in two's complement, even if it is not represented that way internally.

(bitwise-arithmetic-shift #b10000 -3) ⇒ #b00010
Thus, `bitwise-arithmetic-shift` behaves as if defined as follows.

```scheme
(define bitwise-arithmetic-shift
  (lambda (exint1 exint2)
    (if (< exint2 0)
        (bitwise-arithmetic-shift-right exint1 (- exint2))
        (bitwise-arithmetic-shift-left exint1 exint2))))
```

**procedure:** `(bitwise-rotate-bit-field exint1 exint2 exint3 exint4)`
**returns:** see below
**libraries:** `(rnrs arithmetic bitwise), (rnrs)`

`exint2`, `exint3`, and `exint4` must be nonnegative, and `exint2` must not be greater than `exint3`. This procedure returns the result of shifting the bits of `exint1` from bit `exint2` (inclusive) through bit `exint3` (exclusive) left by `(mod exint4 (- exint3 exint2))` bits, with the bits shifted out of the range inserted at the bottom end of the range. `exint1` is treated as if represented in two's complement, even if it is not represented that way internally.

```scheme
(bitwise-rotate-bit-field #b00011010 0 5 3) ⇒ #b00010110
(bitwise-rotate-bit-field #b01101011 2 7 3) ⇒ #b01011011
```

**procedure:** `(bitwise-reverse-bit-field exint1 exint2 exint3)`
**returns:** see below
**libraries:** `(rnrs arithmetic bitwise), (rnrs)`

`exint2` and `exint3` must be nonnegative, and `exint2` must not be greater than `exint3`. This procedure returns the result of reversing the bits of `exint1` from bit `exint2` (inclusive) through bit `exint3` (exclusive). `exint1` is treated as if represented in two's complement, even if it is not represented that way internally.

```scheme
(bitwise-reverse-bit-field #b00011010 0 5) ⇒ #b00001011
(bitwise-reverse-bit-field #b01101011 2 7) ⇒ #b00101111
```

**procedure:** `(string->number string)`
**procedure:** `(string->number string radix)`
**returns:** the number represented by `string`, or `#f`
**libraries:** `(rnrs base), (rnrs)`

If `string` is a valid representation of a number, that number is returned, otherwise `#f` is returned. The number is interpreted in radix `radix`, which must be an exact integer in the set `{2,8,10,16}`. If not specified, `radix` defaults to 10. Any radix specifier within `string`, e.g., `#x`, overrides the `radix` argument.

```scheme
(string->number "0") ⇒ 0
(string->number "3.4e3") ⇒ 3400.0
(string->number "#x#e-2e2") ⇒ -738
(string->number "#e-2e2" 16) ⇒ -738
(string->number "#i15/16") ⇒ 0.9375
(string->number "10" 16) ⇒ 16
```

Section 6.4. Numbers
The Scheme Programming Language, 4th Edition

**procedure:** (number->string num)
**procedure:** (number->string num radix)
**procedure:** (number->string num radix precision)
**returns:** an external representation of num as a string
**libraries:** (rnrs base), (rnrs)

The num is expressed in radix radix, which must be an exact integer in the set \{2,8,10,16\}. If not specified, radix defaults to 10. In any case, no radix specifier appears in the resulting string.

The external representation is such that, when converted back into a number using string->number, the resulting numeric value is equivalent to num. That is, for all inputs:

```scheme
(eqv? (string->number (number->string num radix) radix) num)
```

returns #t. An exception with condition type &implementation-restriction is raised if this is not possible.

If precision is provided, it must be an exact positive integer, num must be inexact, and radix must be 10. In this case, the real part and, if present, the imaginary part of the number are each printed with an explicit mantissa width \(m\), where \(m\) is the least possible value greater than or equal to precision that makes the expression above true.

If radix is 10, inexact values of num are expressed using the fewest number of significant digits possible without violating the above restriction.

```scheme
(number->string 3.4) => "3.4"
(number->string 1e2) => "100.0"
(number->string 1e-23) => "1e-23"
(number->string -7/2) => "-7/2"
(number->string 220/9 16) => "DC/9"
```

### Section 6.5. Fixnums

**Fixnums** represent exact integers in the fixnum range, which is required to be a closed range \([-2^{w-1}, 2^{w-2}-1]\], where \(w\) (the fixnum width) is at least 24. The implementation-specific value of \(w\) may be determined via the procedure fixnum-width, and the endpoints of the range may be determined via the procedures least-fixnum and greatest-fixnum.

The names of arithmetic procedures that operate only on fixnums begin with the prefix "fx" to set them apart from their generic counterparts.

Procedure arguments required to be fixnums are named \(fx\), possibly with a suffix, e.g., \(fx_2\).

Unless otherwise specified, the numeric values of fixnum-specific procedures are fixnums. If the value of a fixnum operation should be a fixnum, but the mathematical result would be outside the fixnum range, an exception with condition type &implementation-restriction is raised.

Bit and shift operations on fixnums assume that fixnums are represented in two's complement, even if they are not represented that way internally.
procedure: (fixnum? obj)
returns: #t if obj is a fixnum, #f otherwise
libraries: (rnrs arithmetic fixnums), (rnrs)

(fixnum? 0) ⇒ #t
(fixnum? -1) ⇒ #t
(fixnum? (- (expt 2 23)) ) ⇒ #t
(fixnum? (- (expt 2 23) 1)) ⇒ #t

procedure: (least-fixnum)
returns: the least (most negative) fixnum supported by the implementation
procedure: (greatest-fixnum)
returns: the greatest (most positive) fixnum supported by the implementation
libraries: (rnrs arithmetic fixnums), (rnrs)

(fixnum? (- (least-fixnum) 1)) ⇒ #f
(fixnum? (least-fixnum)) ⇒ #t
(fixnum? (greatest-fixnum)) ⇒ #t
(fixnum? (+ (greatest-fixnum) 1)) ⇒ #f

procedure: (fixnum-width)
returns: the implementation-dependent fixnum width
libraries: (rnrs arithmetic fixnums), (rnrs)

As described in the lead-in to this section, the fixnum width determines the size of the fixnum range and must be at least 24.

(define w (fixnum-width))
(= (least-fixnum) (- (expt 2 (- w 1)))) ⇒ #t
(= (greatest-fixnum) (- (expt 2 (- w 1)) 1)) ⇒ #t
(>= w 24) ⇒ #t

procedure: (fx=? fx1 fx2 fx3 ...)
procedure: (fx<? fx1 fx2 fx3 ...)
procedure: (fx>? fx1 fx2 fx3 ...)
procedure: (fx<=? fx1 fx2 fx3 ...)
procedure: (fx>=? fx1 fx2 fx3 ...)
returns: #t if the relation holds, #f otherwise
libraries: (rnrs arithmetic fixnums), (rnrs)

The predicate fx=? returns #t if its arguments are equal. The predicate fx<? returns #t if its arguments are monotonically increasing, i.e., each argument is greater than the preceding ones, while fx>? returns #t if its arguments are monotonically decreasing. The predicate fx<=? returns #t if its arguments are monotonically nondecreasing, i.e., each argument is not less than the preceding ones, while fx>=? returns #t if its arguments are monotonically nonincreasing.

(fx=? 0 0) ⇒ #t
(fx=? -1 1) ⇒ #f
(fx<? (least-fixnum) 0 (greatest-fixnum)) ⇒ #t
(let ([x 3]) (fx<=? 0 x 9)) ⇒ #t
(fx>? 5 4 3 2 1) ⇒ #t
(fx<=? 1 3 2) ⇒ #f
(fx>=? 0 0 (least-fixnum)) ⇒ #t
procedure: (fxzero? fx)
returns: #t if fx is zero, #f otherwise

procedure: (fxpositive? fx)
returns: #t if fx is greater than zero, #f otherwise

procedure: (fxnegative? fx)
returns: #t if fx is less than zero, #f otherwise

libraries: (rnrs arithmetic fixnums), (rnrs)

fxzero? is equivalent to (lambda (x) (fx=? x 0)), fxpositive? is equivalent to (lambda (x) (fx>? x 0)), and fxnegative? to (lambda (x) (fx<? x 0)).

(fxzero? 0) ⇒ #t
(fxzero? 1) ⇒ #f
(fxzero? 128) ⇒ #t
(fxzero? 0) ⇒ #f
(fxzero? -1) ⇒ #f

(fxnegative? -65) ⇒ #t
(fxnegative? 0) ⇒ #f
(fxnegative? 1) ⇒ #f

procedure: (fxeven? fx)
returns: #t if fx is even, #f otherwise

procedure: (fxodd? fx)
returns: #t if fx is odd, #f otherwise

libraries: (rnrs arithmetic fixnums), (rnrs)

(fxeven? 0) ⇒ #t
(fxeven? 1) ⇒ #f
(fxeven? -1) ⇒ #f
(fxeven? -10) ⇒ #t

(fxodd? 0) ⇒ #f
(fxodd? 1) ⇒ #t
(fxodd? -1) ⇒ #t
(fxodd? -10) ⇒ #f

procedure: (fxmin fx₁ fx₂ ...)
returns: the minimum of fx₁, fx₂...

procedure: (fxmax fx₁ fx₂ ...)
returns: the maximum of fx₁, fx₂...

libraries: (rnrs arithmetic fixnums), (rnrs)

(fxmin 4 -7 2 0 -6) ⇒ -7
(let ([ls '(7 3 5 2 9 8)])
  (apply fxmin ls)) ⇒ 2
(fxmax 4 -7 2 0 -6) ⇒ 4
(let ([ls '(7 3 5 2 9 8)])
  (apply fxmax ls)) ⇒ 9
procedure: \((\text{fx+ \ fx_1 \ fx_2})\)  
returns: the sum of \(\text{fx}_1\) and \(\text{fx}_2\)  
libraries: \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\((\text{fx+ -3 4}) \Rightarrow 1\)

procedure: \((\text{fx- \ fx})\)  
returns: the additive inverse of \(\text{fx}\)  
procedure: \((\text{fx- \ fx_1 \ fx_2})\)  
returns: the difference between \(\text{fx}_1\) and \(\text{fx}_2\)  
libraries: \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\((\text{fx- 3}) \Rightarrow -3\)  
\((\text{fx- -3 4}) \Rightarrow -7\)

procedure: \((\text{fx* \ fx_1 \ fx_2})\)  
returns: the product of \(\text{fx}_1\) and \(\text{fx}_2\)  
libraries: \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\((\text{fx* -3 4}) \Rightarrow -12\)

procedure: \((\text{fxdiv \ fx_1 \ fx_2})\)  
procedure: \((\text{fxmod \ fx_1 \ fx_2})\)  
procedure: \((\text{fxdiv-and-mod \ fx_1 \ fx_2})\)  
returns: see below  
libraries: \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\(\text{fx}_2\) must not be zero. These are fixnum-specific versions of the generic \(\text{div}, \text{mod}\), and \(\text{div-and-mod}\).

\((\text{fxdiv 17 3}) \Rightarrow 5\)  
\((\text{fxmod 17 3}) \Rightarrow 2\)  
\((\text{fxdiv -17 3}) \Rightarrow -6\)  
\((\text{fxmod -17 3}) \Rightarrow 1\)  
\((\text{fxdiv 17 -3}) \Rightarrow -5\)  
\((\text{fxmod 17 -3}) \Rightarrow 2\)  
\((\text{fxdiv -17 -3}) \Rightarrow 6\)  
\((\text{fxmod -17 -3}) \Rightarrow 1\)

\((\text{fxdiv-and-mod 17 3}) \Rightarrow 5\)

procedure: \((\text{fxdiv0 \ fx_1 \ fx_2})\)  
procedure: \((\text{fxmod0 \ fx_1 \ fx_2})\)  
procedure: \((\text{fxdiv0-and-mod0 \ fx_1 \ fx_2})\)  
returns: see below  
libraries: \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\(\text{fx}_2\) must not be zero. These are fixnum-specific versions of the generic \(\text{div0}, \text{mod0}\), and \(\text{div0-and-mod0}\).

\((\text{fxdiv0 17 3}) \Rightarrow 6\)  
\((\text{fxmod0 17 3}) \Rightarrow -1\)  
\((\text{fxdiv0 -17 3}) \Rightarrow -6\)  
\((\text{fxmod0 -17 3}) \Rightarrow 1\)
When an ordinary fixnum addition, subtraction, or multiplication operation overflows, an exception is raised. These alternative procedures instead return a carry and also allow the carry to be propagated to the next operation. They can be used to implement portable code for multiple-precision arithmetic.

These procedures return the two fixnum values of the following computations. For \( \text{fx+}/\text{carry} \):

\[
(\text{let*} \ ((s \ (+ \ fx_1 \ fx_2 \ fx_3)) \\
\quad [s0 \ (\text{mod0} \ s \ (\text{expt} 2 \ (\text{fixnum-width}))))] \\
\quad [s1 \ (\text{div0} \ s \ (\text{expt} 2 \ (\text{fixnum-width}))))]) \\
\quad (\text{values} \ s0 \ s1))
\]

for \( \text{fx-/carry} \):

\[
(\text{let*} \ ((d \ (- \ fx_1 \ fx_2 \ fx_3)) \\
\quad [d0 \ (\text{mod0} \ d \ (\text{expt} 2 \ (\text{fixnum-width}))))] \\
\quad [d1 \ (\text{div0} \ d \ (\text{expt} 2 \ (\text{fixnum-width}))))]) \\
\quad (\text{values} \ d0 \ d1))
\]

and for \( \text{fx*/carry} \):

\[
(\text{let*} \ ((s \ (+ \ (* \ fx_1 \ fx_2) \ fx_3)) \\
\quad [s0 \ (\text{mod0} \ s \ (\text{expt} 2 \ (\text{fixnum-width}))))] \\
\quad [s1 \ (\text{div0} \ s \ (\text{expt} 2 \ (\text{fixnum-width}))))]) \\
\quad (\text{values} \ s0 \ s1))
\]

procedure: \( \text{fxnot} \ fx \)
returns: the bitwise not of \( fx \)
procedure: \( \text{fxand} \ fx \ldots \)
returns: the bitwise and of \( fx \ldots \)
procedure: \( \text{fxior} \ fx \ldots \)
returns: the bitwise inclusive or of \( fx \ldots \)
procedure: \( \text{fxxor} \ fx \ldots \)
returns: the bitwise exclusive or of \( fx \ldots \)
libraries: (rnrs arithmetic fixnums), (rnrs)
The Scheme Programming Language, 4th Edition

(procedure: (fxif fx1 fx2 fx3)
returns: the bitwise "if" of its arguments
libraries: (rnrs arithmetic fixnums), (rnrs)

For each bit set in \( fx_1 \), the corresponding bit of the result is taken from \( fx_2 \), and for each bit not set in \( fx_1 \), the corresponding bit of the result is taken from \( x_3 \).

\[
(fxif \#b101010 \#b111000 \#b001100) \Rightarrow \#b101100
\]

\( fxif \) might be defined as follows:

\[
(define fxif
  (lambda (fx1 fx2 fx3)
    (fxior (fxand fx1 fx2)
       (fxand (fxnot fx1) fx3))))
\]

(procedure: (fxbit-count fx)
returns: see below
libraries: (rnrs arithmetic fixnums), (rnrs)

For nonnegative inputs, \( fxbit-count \) returns the number of bits set in the two's complement representation of \( fx \). For negative inputs, it returns a negative number whose magnitude is one greater than the number of bits not set in \( fx \), which is equivalent to \( (fxnot \ (fxbit-count \ (fxnot \ fx))) \).

\[
(fxbit-count \#b00000) \Rightarrow 0
\]
\[
(fxbit-count \#b00001) \Rightarrow 1
\]
\[
(fxbit-count \#b00100) \Rightarrow 1
\]
\[
(fxbit-count \#b10101) \Rightarrow 3
\]
\[
(fxbit-count -1) \Rightarrow -1
\]
\[
(fxbit-count -2) \Rightarrow -2
\]
\[
(fxbit-count -4) \Rightarrow -3
\]

(procedure: (fxlength fx)
returns: see below
libraries: (rnrs arithmetic fixnums), (rnrs)

This procedure returns the number of bits of the smallest two's complement representation of \( fx \), not including the sign bit for negative numbers. For 0 \( fxlength \) returns 0.

\[
(fxlength \#b00000) \Rightarrow 0
\]
\[
(fxlength \#b00001) \Rightarrow 1
\]
\[
(fxlength \#b00100) \Rightarrow 3
\]
\[
(fxlength \#b00110) \Rightarrow 3
\]
\[
(fxlength -1) \Rightarrow 0
\]
\[
(fxlength -6) \Rightarrow 3
\]
\[
(fxlength -9) \Rightarrow 4
\]

(procedure: (fxfirst-bit-set fx)
returns: the index of the least significant bit set in \( fx \)
libraries: (rnrs arithmetic fixnums), (rnrs)
If $f_x$ is 0, $f_x$first-bit-set returns -1.

$$(fxfirst-bit-set \ #b00000) \Rightarrow -1$$
$$(fxfirst-bit-set \ #b00001) \Rightarrow 0$$
$$(fxfirst-bit-set \ #b01100) \Rightarrow 2$$

$$(fxfirst-bit-set \ -1) \Rightarrow 0$$
$$(fxfirst-bit-set \ -2) \Rightarrow 1$$
$$(fxfirst-bit-set \ -3) \Rightarrow 0$$

**procedure:** $(fxbit-set? \ f_x \ f_x)$
**returns:** #t if bit $f_x$ of $f_x$ is set, #f otherwise
**libraries:** (rnrs arithmetic fixnums), (rnrs)

$f_x$ must be nonnegative. It is taken as a zero-based index for the bits in the two's complement representation of $f_x$, with the sign bit virtually replicated an infinite number of positions to the left.

$$(fxbit-set? \ #b01011 \ 0) \Rightarrow \#t$$
$$(fxbit-set? \ #b01011 \ 2) \Rightarrow \#f$$

$$(fxbit-set? \ -1 \ 0) \Rightarrow \#t$$
$$(fxbit-set? \ -1 \ 20) \Rightarrow \#t$$
$$(fxbit-set? \ -3 \ 1) \Rightarrow \#f$$
$$(fxbit-set? \ 0 \ (- (fixnum-width) \ 1)) \Rightarrow \#f$$
$$(fxbit-set? \ -1 \ (- (fixnum-width) \ 1)) \Rightarrow \#t$$

**procedure:** $(fxcopy-bit \ f_x \ f_x \ f_x)$
**returns:** $f_x$ with bit $f_x$ replaced by $f_x$3
**libraries:** (rnrs arithmetic fixnums), (rnrs)

$f_x$ must be nonnegative and less than the value of $(- (fixnum-width) \ 1)$. $f_x$ must be 0 or 1. This procedure effectively clears or sets the specified bit depending on the value of $f_x$3.

$$(fxcopy-bit \ #b01110 \ 0 \ 1) \Rightarrow \#b01111$$
$$(fxcopy-bit \ #b01110 \ 2 \ 0) \Rightarrow \#b01010$$

**procedure:** $(fxbit-field \ f_x \ f_x \ f_x)$
**returns:** see below
**libraries:** (rnrs arithmetic fixnums), (rnrs)

$f_x$2 and $f_x$3 must be nonnegative and less than the value of $(fixnum-width)$, and $f_x$2 must not be greater than $f_x$3. This procedure returns the number represented by extracting from $f_x$1 the sequence of bits from $f_x$2 (inclusive) to $f_x$3 (exclusive).

$$(fxbit-field \ #b10110 \ 0 \ 3) \Rightarrow \#b00110$$
$$(fxbit-field \ #b10110 \ 1 \ 3) \Rightarrow \#b00011$$
$$(fxbit-field \ #b10110 \ 2 \ 3) \Rightarrow \#b00001$$
$$(fxbit-field \ #b10110 \ 3 \ 3) \Rightarrow \#b00000$$

**procedure:** $(fxcopy-bit-field \ f_x \ f_x \ f_x \ f_x)$
**returns:** see below
**libraries:** (rnrs arithmetic fixnums), (rnrs)

$f_x$2 and $f_x$3 must be nonnegative and less than the value of $(fixnum-width)$, and $f_x$2 must not be
greater than \( f_x \). This procedure returns \( f_x \) with \( n \) bits from \( f_x \) (inclusive) to \( f_x \) (exclusive) replaced by the low-order \( n \) bits of \( x' \).

\[
\text{fxcopy-bit-field} \ #b10000 \ 0 \ 3 \ #b10101) \Rightarrow \ #b10101
\]
\[
\text{fxcopy-bit-field} \ #b10000 \ 1 \ 3 \ #b10101) \Rightarrow \ #b10010
\]
\[
\text{fxcopy-bit-field} \ #b10000 \ 2 \ 3 \ #b10101) \Rightarrow \ #b10100
\]
\[
\text{fxcopy-bit-field} \ #b10000 \ 3 \ 3 \ #b10101) \Rightarrow \ #b10000
\]

**procedure:** \((\text{fxarithmetic-shift-right} \ f_x \ f_x)\)

**returns:** \( f_x \) arithmetically shifted right by \( f_x \) bits

**procedure:** \((\text{fxarithmetic-shift-left} \ f_x \ f_x)\)

**returns:** \( f_x \) shifted left by \( f_x \) bits

**libraries:** \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\( f_x \) must be nonnegative and less than the value of \((\text{fixnum-width})\).

\[
\text{fxarithmetic-shift-right} \ #b10000 \ 3) \Rightarrow \ #b00010
\]
\[
\text{fxarithmetic-shift-right} \ -1 \ 1) \Rightarrow \ -1
\]
\[
\text{fxarithmetic-shift-right} \ -64 \ 3) \Rightarrow \ -8
\]
\[
\text{fxarithmetic-shift-left} \ #b00010 \ 2) \Rightarrow \ #b01000
\]
\[
\text{fxarithmetic-shift-left} \ -1 \ 2) \Rightarrow \ -4
\]

**procedure:** \((\text{fxarithmetic-shift} \ f_x \ f_x)\)

**returns:** see below

**libraries:** \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

The absolute value of \( f_x \) must be less than the value of \((\text{fixnum-width})\). If \( f_x \) is negative, \( \text{fxarithmetic-shift} \) returns the result of arithmetically shifting \( f_x \) right by \( f_x \) bits. Otherwise, \( \text{fxarithmetic-shift} \) returns the result of shifting \( f_x \) left by \( f_x \) bits.

\[
\text{fxarithmetic-shift} \ #b10000 \ -3) \Rightarrow \ #b00010
\]
\[
\text{fxarithmetic-shift} \ -1 \ -1) \Rightarrow \ -1
\]
\[
\text{fxarithmetic-shift} \ -64 \ -3) \Rightarrow \ -8
\]
\[
\text{fxarithmetic-shift} \ #b00010 \ 2) \Rightarrow \ #b01000
\]
\[
\text{fxarithmetic-shift} \ -1 \ 2) \Rightarrow \ -4
\]

Thus, \( \text{fxarithmetic-shift} \) behaves as if defined as follows.

\[
\text{define fxarithmetic-shift}
\quad \quad \text{(lambda} \ (fx1 fx2)
\quad \quad \quad \text{(if} \ (fx<\? \ fx2 \ 0)
\quad \quad \quad \quad \text{(fxarithmetic-shift-right} \ fx1 \ (fx- \ fx2))
\quad \quad \quad \quad \text{(fxarithmetic-shift-left} \ fx1 \ fx2)))
\]

**procedure:** \((\text{fxrotate-bit-field} \ f_x \ f_x \ f_x \ f_x)\)

**returns:** see below

**libraries:** \((\text{rnrs arithmetic fixnums}), (\text{rnrs})\)

\( f_x \) and \( f_x \) and \( f_x \) and \( f_x \) must be nonnegative and less than the value of \((\text{fixnum-width})\), \( f_x \) must not be greater than \( f_x \), and \( f_x \) must not be greater than the difference between \( f_x \) and \( f_x \).

This procedure returns the result of shifting the bits of \( f_x \) from bit \( f_x \) (inclusive) through bit \( f_x \) (exclusive) left by \( f_x \) bits, with the bits shifted out of the range inserted at the bottom end of the range.
procedure: (fxrotate-bit-field fx1 fx2 fx3)
returns: see below
libraries: (rnrs arithmetic fixnums), (rnrs)

fx2 and fx3 must be nonnegative and less than the value of (fixnum-width), and fx2 must not be greater than fx3. This procedure returns the result of reversing the bits of fx1 from bit fx2 (inclusive) through bit fx3 (exclusive).

(fxreverse-bit-field #b00011010 0 5) ⇒ #b00001011
(fxreverse-bit-field #b01101011 2 7) ⇒ #b00101111

Section 6.6. Flonums

Flonums represent inexact real numbers. Implementations are required to represent as a flonum any inexact real number whose lexical syntax contains no vertical bar and no exponent marker other than e, but are not required to represent any other inexact real number as a flonum.

Implementations typically use the IEEE double-precision floating-point representation for flonums, but implementations are not required to do so or even to use a floating-point representation of any sort, despite the name "flonum."

This section describes operations on flonums. Flonum-specific procedure names begin with the prefix "fl" to set them apart from their generic counterparts.

Procedure arguments required to be flonums are named fl, possibly with suffix, e.g., fl2. Unless otherwise specified, the numeric values of flonum-specific procedures are flonums.

procedure: (flonum? obj)
returns: #t if obj is a flonum, otherwise #f
libraries: (rnrs arithmetic flonums), (rnrs)

(flonum? 0) ⇒ #f
(flonum? 3/4) ⇒ #f
(flonum? 3.5) ⇒ #t
(flonum? .02) ⇒ #t
(flonum? 1e10) ⇒ #t
(flonum? 3.0+0.0i) ⇒ #f

Procedure: (fl=? fl1 fl2 fl3 ...)
Procedure: (fl<? fl1 fl2 fl3 ...)
Procedure: (fl>? fl1 fl2 fl3 ...)
Procedure: (fl<=? fl1 fl2 fl3 ...)
Procedure: (fl>=? fl1 fl2 fl3 ...)
returns: #t if the relation holds, #f otherwise
libraries: (rnrs arithmetic flonums), (rnrs)

The predicate fl=? returns #t if its arguments are equal. The predicate fl<? returns #t if its arguments are monotonically increasing, i.e., each argument is greater than the preceding ones, while fl>? returns #t if its arguments are monotonically decreasing. The predicate fl<=? returns #t if its arguments are monotonically nondecreasing, i.e., each argument is not less than the preceding ones, while fl>=? returns #t if its arguments are monotonically nonincreasing. When passed only one argument, each of these predicates returns #t.
Comparisons involving NaNs always return \#f.

\[
\begin{align*}
(f1=? 0.0 0.0) &\Rightarrow \#t \\
(f1<? -1.0 0.0 1.0) &\Rightarrow \#t \\
(f1>? -1.0 0.0 1.0) &\Rightarrow \#f \\
(f1<=? 0.0 3.0 3.0) &\Rightarrow \#t \\
(f1>=? 4.0 3.0 3.0) &\Rightarrow \#t \\
(f1>? 7.0 +inf.0) &\Rightarrow \#t \\
(f1=? +nan.0 0.0) &\Rightarrow \#f \\
(f1=? +nan.0 +nan.0) &\Rightarrow \#f \\
(f1<? +nan.0 +nan.0) &\Rightarrow \#f \\
(f1<=? +nan.0 +inf.0) &\Rightarrow \#f \\
(f1>=? +nan.0 +inf.0) &\Rightarrow \#f \\
\end{align*}
\]

**procedure**: \( (flzero? \ f1) \)
**returns**: \#t if \( \ f1 \) is zero, \#f otherwise

**procedure**: \( (flpositive? \ f1) \)
**returns**: \#t if \( \ f1 \) is greater than zero, \#f otherwise

**procedure**: \( (flnegative? \ f1) \)
**returns**: \#t if \( \ f1 \) is less than zero, \#f otherwise

**libraries**: \((rnrs\ arithmetic\ flonums),(rnrs)\)

\( flzero? \) is equivalent to \((\text{lambda} \ (x) \ (fl=? x 0.0))\), \( flpositive? \) is equivalent to \((\text{lambda} \ (x) \ (fl>? x 0.0))\), and \( flnegative? \) to \((\text{lambda} \ (x) \ (fl<? x 0.0))\).

Even if the flonum representation distinguishes -0.0 from +0.0, -0.0 is considered both zero and nonnegative.

\[
\begin{align*}
(flzero? 0.0) &\Rightarrow \#t \\
(flzero? 1.0) &\Rightarrow \#f \\
(flpositive? 128.0) &\Rightarrow \#t \\
(flpositive? 0.0) &\Rightarrow \#f \\
(flpositive? -1.0) &\Rightarrow \#f \\
(flnegative? -65.0) &\Rightarrow \#t \\
(flnegative? 0.0) &\Rightarrow \#f \\
(flnegative? 1.0) &\Rightarrow \#f \\
(flnegative? -0.0) &\Rightarrow \#t \\
(flnegative? -0.0) &\Rightarrow \#f \\
(flnegative? +nan.0) &\Rightarrow \#f \\
(flnegative? +nan.0) &\Rightarrow \#f \\
(flnegative? +inf.0) &\Rightarrow \#f \\
(flnegative? -inf.0) &\Rightarrow \#t \\
\end{align*}
\]

**procedure**: \( (flinteger? \ f1) \)
**returns**: \#t if \( \ f1 \) is integer, \#f otherwise

**libraries**: \((rnrs\ arithmetic\ flonums),(rnrs)\)
(flinteger? 0.0) => #t
(flinteger? -17.0) => #t
(flinteger? +nan.0) => #f
(flinteger? +inf.0) => #f

procedure: (flfinite? fl)
returns: #t if fl is finite, #f otherwise

procedure: (flinfinite? fl)
returns: #t if fl is infinite, #f otherwise

procedure: (flnan? fl)
returns: #t if fl is a NaN, #f otherwise

libraries: (rnrs arithmetic flonums), (rnrs)

(flfinite? 3.1415) => #t
(flfinite? 3.1415) => #f
(flnan? 3.1415) => #f

(flfinite? +inf.0) => #f
(flfinite? -inf.0) => #t
(flnan? -inf.0) => #f

(flfinite? +nan.0) => #f
(flfinite? +nan.0) => #f
(flnan? +nan.0) => #t

procedure: (fleven? fl-int)
returns: #t if fl-int is even, #f otherwise

procedure: (flodd? fl-int)
returns: #t if fl-int is odd, #f otherwise

libraries: (rnrs arithmetic flonums), (rnrs)

fl-int must be an integer-valued flonum.

(fleven? 0.0) => #t
(fleven? 1.0) => #f
(fleven? -1.0) => #f
(fleven? -10.0) => #t

(flodd? 0.0) => #f
(flodd? 1.0) => #t
(flodd? -1.0) => #t
(flodd? -10.0) => #f

procedure: (flmin fl1 fl2 ...)
returns: the minimum of fl1 fl2 ...

procedure: (flmax fl1 fl2 ...)
returns: the maximum of fl1 fl2 ...

libraries: (rnrs arithmetic flonums), (rnrs)

(flmin 4.2 -7.5 2.0 0.0 -6.4) => -7.5

(let ((ls '(7.1 3.5 5.0 2.6 2.6 8.0)))
  (apply flmin ls)) => 2.6
(flmax 4.2 -7.5 2.0 0.0 -6.4) ⇒ 4.2

(let ([ls '[(7.1 3.5 5.0 2.6 2.6 8.0)]]
  (apply flmax ls)) ⇒ 8.0

**procedure**: (fl+ f1 ...)
**returns**: the sum of the arguments f1 ...
**libraries**: (rnrs arithmetic flonums), (rnrs)

When called with no arguments, fl+ returns 0.0.

(fl+) ⇒ 0.0
(fl+ 1.0 2.5) ⇒ 3.25
(fl+ 3.0 4.25 5.0) ⇒ 12.25
(apply fl+ '(1.0 2.0 3.0 4.0 5.0)) ⇒ 15.0

**procedure**: (fl- f1)
**returns**: the additive inverse of f1
**procedure**: (fl- f1 f2 f3 ...)
**returns**: the difference between f1, and the sum of f1 f2 f3 ...
**libraries**: (rnrs arithmetic flonums), (rnrs)

With an IEEE floating-point representation of flonums, the single-argument fl- is equivalent to

(lambda (x) (fl* -1.0 x))

or

(lambda (x) (fl- -0.0 x))

but not

(lambda (x) (fl- 0.0 x))

since the latter returns 0.0 rather than -0.0 for 0.0.

(fl- 0.0) ⇒ -0.0
(fl- 3.0) ⇒ -3.0
(fl- 4.0 3.0) ⇒ 1.0
(fl- 4.0 3.0 2.0 1.0) ⇒ -2.0

**procedure**: (fl* f1 ...)
**returns**: the product of the arguments f1 ...
**libraries**: (rnrs arithmetic flonums), (rnrs)

When called with no arguments, fl* returns 1.0.

(fl*) ⇒ 1.0
(fl* 1.5 2.5) ⇒ 3.75
(fl* 3.0 -4.0 5.0) ⇒ -60.0
(apply fl* '(1.0 -2.0 3.0 -4.0 5.0)) ⇒ 120.0

**procedure**: (fl/ f1)
**returns**: the multiplicative inverse of f1
procedure: (fl/ fl₁ fl₂ fl₃ ...)
returns: the result of dividing \( fl₁ \) by the product of \( fl₂ \) \( fl₃ \) ...
libraries: (rnrs arithmetic flonums), (rnrs)

\[
(\text{fl/ } -4.0) \Rightarrow -0.25  \\
(\text{fl/ } 8.0 -2.0) \Rightarrow -4.0  \\
(\text{fl/ } -9.0 2.0) \Rightarrow -4.5  \\
(\text{fl/ } 60.0 5.0 3.0 2.0) \Rightarrow 2.0
\]

procedure: (fl/div fl₁ fl₂)
procedure: (fl/mod fl₁ fl₂)
procedure: (fl/div-and-mod fl₁ fl₂)
returns: see below
libraries: (rnrs arithmetic flonums), (rnrs)

These are flonum-specific versions of the generic \( \text{div}, \mod, \) and \( \text{div-and-mod}. \)

\[
(\text{fl/div } 17.0 3.0) \Rightarrow 5.0  \\
(\text{fl/mod } 17.0 3.0) \Rightarrow 2.0  \\
(\text{fl/div } -17.0 3.0) \Rightarrow -6.0  \\
(\text{fl/mod } -17.0 3.0) \Rightarrow 1.0  \\
(\text{fl/div } 17.0 -3.0) \Rightarrow -5.0  \\
(\text{fl/mod } 17.0 -3.0) \Rightarrow 2.0  \\
(\text{fl/div } -17.0 -3.0) \Rightarrow 6.0  \\
(\text{fl/mod } -17.0 -3.0) \Rightarrow 1.0
\]

\[
(\text{fl/div-and-mod } 17.5 3.75) \Rightarrow 4.0  \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 2.5
\]

procedure: (fl/div0 fl₁ fl₂)
procedure: (fl/mod0 fl₁ fl₂)
procedure: (fl/div0-and-mod0 fl₁ fl₂)
returns: see below
libraries: (rnrs arithmetic flonums), (rnrs)

These are flonum-specific versions of the generic \( \text{div0}, \mod0, \) and \( \text{div0-and-mod0}. \)

\[
(\text{fl/div0 } 17.0 3.0) \Rightarrow 6.0  \\
(\text{fl/mod0 } 17.0 3.0) \Rightarrow -1.0  \\
(\text{fl/div0 } -17.0 3.0) \Rightarrow -6.0  \\
(\text{fl/mod0 } -17.0 3.0) \Rightarrow 1.0  \\
(\text{fl/div0 } 17.0 -3.0) \Rightarrow -6.0  \\
(\text{fl/mod0 } 17.0 -3.0) \Rightarrow -1.0  \\
(\text{fl/div0 } -17.0 -3.0) \Rightarrow 6.0  \\
(\text{fl/mod0 } -17.0 -3.0) \Rightarrow 1.0
\]

\[
(\text{fl/div0-and-mod0 } 17.5 3.75) \Rightarrow 5.0  \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad -1.25
\]

procedure: (fl/round fl)
returns: the integer closest to \( fl \)
procedure: (fl/truncate fl)
returns: the integer closest to \( fl \) toward zero
procedure: (fl/floor fl)
returns: the integer closest to fl toward \(-\infty\)
procedure: (flceiling fl)
returns: the integer closest to fl toward \(+\infty\)
libraries: (rnrs arithmetic flonums), (rnrs)

If fl is an integer, NaN, or infinity, each of these procedures returns fl. If fl is exactly between two integers, flround returns the closest even integer.

(flround 17.3) \Rightarrow 17.0
(flround -17.3) \Rightarrow -17.0
(flround 2.5) \Rightarrow 2.0
(flround 3.5) \Rightarrow 4.0

(fltruncate 17.3) \Rightarrow 17.0
(fltruncate -17.3) \Rightarrow -17.0

(flfloor 17.3) \Rightarrow 17.0
(flfloor -17.3) \Rightarrow -18.0

(flceiling 17.3) \Rightarrow 18.0
(flceiling -17.3) \Rightarrow -17.0

procedure: (flnumerator fl)
returns: the numerator of fl

procedure: (fldenominator fl)
returns: the denominator of fl

libraries: (rnrs arithmetic flonums), (rnrs)

If fl is an integer, including 0.0, or infinity, the numerator is fl and the denominator is 1.0.

(flnumerator -9.0) \Rightarrow -9.0
(fldenominator -9.0) \Rightarrow 1.0
(flnumerator 0.0) \Rightarrow 0.0
(fldenominator 0.0) \Rightarrow 1.0
(flnumerator -inf.0) \Rightarrow -inf.0
(fldenominator -inf.0) \Rightarrow 1.0

The following hold for IEEE floats, but not necessarily other flonum representations.

(flnumerator 3.5) \Rightarrow 7.0
(fldenominator 3.5) \Rightarrow 2.0

procedure: (flabs fl)
returns: absolute value of fl

libraries: (rnrs arithmetic flonums), (rnrs)

(flabs 3.2) \Rightarrow 3.2
(flabs -2e-20) \Rightarrow 2e-20

procedure: (flexp fl)
returns: \(e\) to the fl power

procedure: (fllog fl)
returns: the natural logarithm of fl

procedure: (fllog fl1, fl2)
returns: the base-$f_2$ logarithm of $f_1$

libraries: (rnrs arithmetic flonums), (rnrs)

(flexp 0.0) → 1.0
(flexp 1.0) → 2.7182818284590455

(fllog 1.0) → 0.0
(fllog (exp 1.0)) → 1.0
(fl/ (fllog 100.0) (fllog 10.0)) → 2.0

(fllog 100.0 10.0) → 2.0
(fllog .125 2.0) → -3.0

procedure: (fllsin $f_1$)
returns: the sine of $f_1$

procedure: (flcos $f_1$)
returns: the cosine of $f_1$

procedure: (fltan $f_1$)
returns: the tangent of $f_1$

libraries: (rnrs arithmetic flonums), (rnrs)

procedure: (flasin $f_1$)
returns: the arc sine of $f_1$

procedure: (flacos $f_1$)
returns: the arc cosine of $f_1$

procedure: (flatan $f_1$)
returns: the arc tangent of $f_1$

procedure: (flatan $f_1$, $f_2$)
returns: the arc tangent of $f_1/f_2$

libraries: (rnrs arithmetic flonums), (rnrs)

procedure: (flsqrt $f_1$)
returns: the principal square root of $f_1$

libraries: (rnrs arithmetic flonums), (rnrs)

Returns the principal square root of $f_1$. The square root of -0.0 should be -0.0. The result for other negative numbers may be a NaN or some other unspecified flonum.

(flsqrt 4.0) → 2.0
(flsqrt 0.0) → 0.0
(flsqrt -0.0) → -0.0

procedure: (flexpt $f_1$, $f_2$)
returns: $f_1$ raised to the $f_2$ power

libraries: (rnrs arithmetic flonums), (rnrs)

If $f_1$ is negative and $f_2$ is not an integer, the result may be a NaN or some other unspecified flonum. If $f_1$ and $f_2$ are both zero, the result is 1.0. If $f_1$ is zero and $f_2$ is positive, the result is zero. In other cases where $f_1$ is zero, the result may be a NaN or some other unspecified flonum.

(flexpt 3.0 2.0) → 9.0
(flexpt 0.0 +inf.0) → 0.0

procedure: (fixnum->flonum $fx$)
returns: the flonum representation closest to \( fx \)

**procedure**: `(real->flonum real)`

returns: the flonum representation closest to `real`

**libraries**: `(rnrs arithmetic flonums), (rnrs)`

fixnum->flonum is a restricted variant of inexact. `real->flonum` is a restricted variant of inexact when the input is an exact real; when it is an inexact non-flonum real, it converts the inexact non-flonum real into the closest flonum.

```
(fixnum->flonum 0) ⇒ 0.0
(fixnum->flonum 13) ⇒ 13.0
(real->flonum -1/2) ⇒ -0.5
(real->flonum 1s3) ⇒ 1000.0
```

**Section 6.7. Characters**

Characters are atomic objects representing letters, digits, special symbols such as \( $ \) or \( - \), and certain non-graphic control characters such as space and newline. Characters are written with a `#\` prefix. For most characters, the prefix is followed by the character itself. The written character representation of the letter \( \Lambda \), for example, is `#\Lambda`. The characters newline, space, and tab may be written in this manner as `#\newline`, `#\space`, and `#\tab`. Other character names are supported as well, as defined by the grammar for character objects on page 457. Any Unicode character may be written with the syntax `#\xn`, where \( n \) consists of one or more hexadecimal digits and represents a valid Unicode scalar value.

This section describes the operations that deal primarily with characters. See also the following section on strings and Chapter 7 on input and output for other operations relating to characters.

**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**: `(char=>? char_1 char_2 char_3 ...)`

returns: `#t` if the relation holds, `#f` otherwise

**libraries**: `(rnrs base), (rnrs)`

These predicates behave in a similar manner to the numeric predicates `=`, `<`, `>`, `<=`, and `>=`. For example, `char=?` returns `#t` when its arguments are equivalent characters, and `char<?` returns `#t` when its arguments are monotonically increasing character (Unicode scalar) values.

```
(char>? #\a #\b) ⇒ #f
(char<? #\a #\b) ⇒ #t
(char<? #\a #\b #\c) ⇒ #t
(let ([c #\r])
  (char<=? #\a c #\z)) ⇒ #t
(char<=? #\Z #\W) ⇒ #f
(char=? #\+ #\+) ⇒ #t
```

**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**: `(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 ...)`

Section 6.7. Characters 167
returns: #t if the relation holds, #f otherwise
libraries: (rnrs unicode), (rnrs)

These predicates are identical to the predicates char=?, char<?, char>? , char<=?, and char=>?
except that they are case-insensitive, i.e., compare the case-folded versions of their arguments. For example,
char=? considers #\a and #\A to be distinct values; char-ci=? does not.

(char-ci<? #\a #\B) ⇒ #t
(char-ci=? #\W #\w) ⇒ #t
(char-ci=? #\= #\+) ⇒ #f
(let ([c #\R])
  (list (char-ci<=? #\a c #\z)
       (char-ci=? #\a c #\z))) ⇒ (#f #t)

procedure: (char-alphabetic? char)
returns: #t if char is a letter, #f otherwise

procedure: (char-numeric? char)
returns: #t if char is a digit, #f otherwise

procedure: (char-whitespace? char)
returns: #t if char is whitespace, #f otherwise
libraries: (rnrs unicode), (rnrs)

A character is alphabetic if it has the Unicode "Alphabetic" property, numeric if it has the Unicode "Numeric"
property, and whitespace if has the Unicode "White_Space" property.

(char-alphabetic? #\a) ⇒ #t
(char-alphabetic? #\T) ⇒ #t
(char-alphabetic? #\b) ⇒ #f
(char-alphabetic? #\$) ⇒ #f

(char-numeric? #\7) ⇒ #t
(char-numeric? #\2) ⇒ #t
(char-numeric? #\X) ⇒ #f
(char-numeric? #\space) ⇒ #f

(char-whitespace? #\space) ⇒ #t
(char-whitespace? #\newline) ⇒ #t
(char-whitespace? #\Z) ⇒ #f

procedure: (char-lower-case? char)
returns: #t if char is lower case, #f otherwise

procedure: (char-upper-case? char)
returns: #t if char is upper case, #f otherwise

procedure: (char-title-case? char)
returns: #t if char is title case, #f otherwise
libraries: (rnrs unicode), (rnrs)

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-lower-case? #\r) ⇒ #t
(char-lower-case? #\R) ⇒ #f
(char-upper-case? #\r) ⇒ #f
procedure: (char-general-category char)
returns: a symbol representing the Unicode general category of char
libraries: (rnrs unicode), (rnrs)

The return value is one of the symbols Lu, Ll, Lt, Lm, Lo, Mn, 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

procedure: (char-upcase char)
returns: the upper-case character counterpart of char
libraries: (rnrs unicode), (rnrs)

If char is a lower- or title-case character and has a single upper-case counterpart, char-upcase returns the upper-case counterpart. Otherwise char-upcase returns char.

(char-upcase \g) ⇒ \G
(char-upcase \G) ⇒ \G
(char-upcase \7) ⇒ \7
(char-upcase \ω) ⇒ \Ω

procedure: (char-downcase char)
returns: the lower-case character equivalent of char
libraries: (rnrs unicode), (rnrs)

If char is an upper- or title-case character and has a single lower-case counterpart, char-downcase returns the lower-case counterpart. Otherwise char-downcase returns char.

(char-downcase \G) ⇒ \g
(char-downcase \g) ⇒ \g
(char-downcase \7) ⇒ \7
(char-downcase \ω) ⇒ \ω

procedure: (char-titlecase char)
returns: the title-case character equivalent of char
libraries: (rnrs unicode), (rnrs)

If char is an upper- or lower-case character and has a single title-case counterpart, char-titlecase returns the title-case counterpart. Otherwise, if it is not a title-case character, has no single title-case counterpart, but does have a single upper-case counterpart, char-titlecase returns the upper-case counterpart. Otherwise char-titlecase returns char.

(char-titlecase \G) ⇒ \G
(char-titlecase \g) ⇒ \G
(char-titlecase \7) ⇒ \7
(char-titlecase \ω) ⇒ \Ω
Section 6.8. Strings

Strings are sequences of characters and are often used as messages, character buffers, or containers for blocks of text. Scheme provides operations for creating strings, extracting characters from strings, obtaining substrings, concatenating strings, and altering the contents of strings.

A string is written as a sequence of characters enclosed in double quotes, e.g., "hi there". A double quote may be introduced into a string by preceding it by a backward slash, e.g., "two \"quotes\" within". A backward slash may also be included by preceding it with a backward slash, e.g., "a \slashes". Various special characters can be inserted with other two-character sequences, e.g., \n for newline, \r for carriage return, and \t for tab. Any Unicode character may be inserted with the syntax \uxxx, where xxx consists of one or more hexadecimal digits and represents a valid Unicode scalar value. A grammar defining the precise syntax of strings is given on page 458.

Strings are indexed by exact nonnegative integers, and the index of the first element of any string is 0. The highest valid index for a given string is one less than its length.

```scheme
 procedure: (string=? string1 string2 string3 ...)
 procedure: (string<? string1 string2 string3 ...)
 procedure: (string>? string1 string2 string3 ...)
 procedure: (string<=? string1 string2 string3 ...)
 procedure: (string=>? string1 string2 string3 ...)
 returns: #t if the relation holds, #f otherwise
```

The Scheme Programming Language, 4th Edition
libraries: (rnrs base), (rnrs)

As with =, <, >, <=, and >=, these predicates express relationships among all of the arguments. For example, string>? determines if the lexicographic ordering of its arguments is monotonically decreasing.

The comparisons are based on the character predicates char=? and char<?. Two strings are lexicographically equivalent if they are the same length and consist of the same sequence of characters according to char=?. If two strings differ only in length, the shorter string is considered to be lexicographically less than the longer string. Otherwise, the first character position at which the strings differ (by char=? or char<?) determines which string is lexicographically less than the other, according to char<?.

Two-argument string=? may be defined without error checks as follows.

```
(define string=?
  (lambda (s1 s2)
    (let ([n (string-length s1)])
      (and (= n (string-length s2))
        (let loop ([i 0])
          (or (= i n)
            (and (char=? (string-ref s1 i) (string-ref s2 i))
              (loop (+ i 1))))))))
```

Two-argument string<? may be defined without error checks as follows.

```
(define string<?
  (lambda (s1 s2)
    (let ([n1 (string-length s1)] [n2 (string-length s2)])
      (let loop ([i 0])
        (and (not (= i n2))
          (or (= i n1)
            (let ([c1 (string-ref s1 i)] [c2 (string-ref s2 i)])
              (or (char<? c1 c2)
                (and (char=? c1 c2)
                  (loop (+ i 1))))))))))
```

These definitions may be extended straightforwardly to support three or more arguments. string<=?, string>? and string>=? may be defined similarly.

```
(string=? "mom" "mom") ⇒ #t
(string<? "mom" "mommy") ⇒ #t
(string>? "Dad" "Dad") ⇒ #f
(string=? "Mom and Dad" "mom and dad") ⇒ #f
(string<? "a" "b" "c") ⇒ #t
```

procedure: (string-ci=? string1 string2 string3 ...)
procedure: (string-ci<? string1 string2 string3 ...)
procedure: (string-ci<=? string1 string2 string3 ...)
procedure: (string-ci=>? string1 string2 string3 ...)
procedure: (string-ci>=? string1 string2 string3 ...)
returns: #t if the relation holds, #f otherwise
libraries: (rnrs unicode), (rnrs)

These predicates are identical to string=?, string<?, string>?, string<=?, and string=>? except that they are case-sensitive, i.e., compare the case-folded versions of their arguments.

Section 6.8. Strings 171
procedure: (string char ...)  
returns: a string containing the characters char ...  
libraries: (rnrs base), (rnrs)

(string) ⇒ ""  
(string \a \b \c) ⇒ "abc"  
(string \H \E \Y \!) ⇒ "HEY!"

procedure: (make-string n)  
procedure: (make-string n char)  
returns: a string of length n  
libraries: (rnrs base), (rnrs)

n must be an exact nonnegative integer. If char is supplied, the string is filled with n occurrences of char, otherwise the characters contained in the string are unspecified.

(make-string 0) ⇒ ""  
(make-string 0 \x) ⇒ ""  
(make-string 5 \x) ⇒ "xxxxx"

procedure: (string-length string)  
returns: the number of characters in string  
libraries: (rnrs base), (rnrs)

The length of a string is always an exact nonnegative integer.

(string-length "abc") ⇒ 3  
(string-length "") ⇒ 0  
(string-length "hi there") ⇒ 8  
(string-length (make-string 1000000)) ⇒ 1000000

procedure: (string-ref string n)  
returns: the nth character (zero-based) of string  
libraries: (rnrs base), (rnrs)

n must be an exact nonnegative integer less than the length of string.

(string-ref "hi there" 0) ⇒ \h  
(string-ref "hi there" 5) ⇒ \e

procedure: (string-set! string n char)  
returns: unspecified  
libraries: (rnrs mutable-strings)

n must be an exact nonnegative integer less than the length of string. string-set! changes the nth element of string to char.

(let ([str (string-copy "hi three")])  
  (string-set! str 5 \e)
procedure: (string-copy string)
returns: a new copy of string
libraries: (rnrs base), (rnrs)

This procedure creates a new string with the same length and contents as string.

(string-copy "abc") ⇒ "abc"

(let ([str "abc"])
  (eq? str (string-copy str))) ⇒ #f

procedure: (string-append string ...)
returns: a new string formed by concatenating the strings string ...
libraries: (rnrs base), (rnrs)

(string-append) ⇒ ""
(string-append "abc" "def") ⇒ "abcdef"
(string-append "Hey " "you " "there!") ⇒ "Hey you there!"

The following implementation of string-append recurs down the list of strings to compute the total length, then allocates the new string, then fills it up as it unwinds the recursion.

(define string-append
  (lambda args
    (let f ([ls args] [n 0])
      (if (null? ls)
        (make-string n)
        (let* ([s1 (car ls)]
          [m (string-length s1)]
          [s2 (f (cdr ls) (+ n m))])
          (do ([i 0 (+ i 1)] [j n (+ j 1)])
            ((= i m) s2)
            (string-set! s2 j (string-ref s1 i)))))))

procedure: (substring string start end)
returns: a copy of string from start (inclusive) to end (exclusive)
libraries: (rnrs base), (rnrs)

start and end must be exact nonnegative integers; start must be less than the length of string, while end may be less than or equal to the length of string. If end ≤ start, a string of length zero is returned. substring may be defined without error checks as follows.

(define substring
  (lambda (s1 m n)
    (let ((s2 (make-string (- n m))))
      (do ([j 0 (+ j 1)] [i m (+ i 1)])
        ((= i n) s2)
        (string-set! s2 j (string-ref s1 i)))))

(substring "hi there" 0 1) ⇒ "h"
(substring "hi there" 3 6) ⇒ "the"
(substring "hi there" 5 5) ⇒ ""

(let ([str "hi there"])
  (let ([end (string-length str)])
    (substring str 0 end))) ⇒ "hi there"

procedure: (string-fill! string char)
returns: unspecified
libraries: (rnrs mutable-strings)

string-fill! sets every character in string to char.

(let ([str (string-copy "sleepy")])
  (string-fill! str #\Z)
  str) ⇒ "ZZZZZZ"

string-fill! might be defined as follows:

(define string-fill!
  (lambda (s c)
    (let ([n (string-length s)])
      (do ([i 0 (+ i 1)])
          ((= i n))
        (string-set! s i c)))))

An alternative definition is given on page 276.

procedure: (string-upcase string)
returns: the upper-case equivalent of string
procedure: (string-downcase string)
returns: the lower-case equivalent of string
procedure: (string-foldcase string)
returns: the case-folded equivalent of string
procedure: (string-titlecase string)
returns: the title-case equivalent of string
libraries: (rnrs unicode), (rnrs)

These procedures implement Unicode's locale-independent case mappings from scalar-value sequences to scalar-value sequences. These mappings do not always map single characters to single characters, so the length of the result string may differ from the length of string. If the result string is the same as string (by string=?), string or a copy of string may be returned. Otherwise, the result string is newly allocated. string-foldcase does not use the special mappings for Turkic languages.

string-titlecase converts the first cased character of each word in string to its title-case counterpart and converts each other character to its lower-case counterpart. Word breaks are recognized as specified in Unicode Standard Annex #29 [8].

(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") \(\Rightarrow\) "strasse"

(string-downcase "\(\Sigma\)") \(\Rightarrow\) "\(\sigma\"

(string-titlecase "kNock KNoCK") \(\Rightarrow\) "Knock Knock"
(string-titlecase "who's there?") \(\Rightarrow\) "Who's There?"
(string-titlecase "r6rs") \(\Rightarrow\) "R6rs"
(string-titlecase "R6RS") \(\Rightarrow\) "R6rs"

procedure: (string-normalize-nfd string)
returns: the Unicode normalized form D of string
procedure: (string-normalize-nfkd string)
returns: the Unicode normalized form KD of string
procedure: (string-normalize-nfc string)
returns: the Unicode normalized form C of string
procedure: (string-normalize-nfkc string)
returns: the Unicode normalized form KC of string

libraries: (rnrs unicode), (rnrs)

If the result string is the same as string (by string=?), string or a copy of string may be returned. Otherwise, the result string is newly allocated.

(string-normalize-nfd "\xE9;") \(\Rightarrow\) "e\x301;"
(string-normalize-nfc "\xE9;") \(\Rightarrow\) "\xE9;"
(string-normalize-nfd "\xE5;\xE9;") \(\Rightarrow\) "e\x301;"
(string-normalize-nfc "\xE5;\xE9;") \(\Rightarrow\) "\xE9;"

procedure: (string->list string)
returns: a list of the characters in string
libraries: (rnrs base), (rnrs)

string->list allows a string to be converted into a list, so that Scheme's list-processing operations may be applied to the processing of strings. string->list may be defined without error checks as follows.

(define string->list
  (lambda (s)
    (do ((i (- (string-length s) 1) (- i 1))
          [ls '() (cons (string-ref s i) ls)]
     ((< i 0) ls))))

(string->list "") \(\Rightarrow\) ()
(string->list "abc") \(\Rightarrow\) (#\a #\b #\c)
(apply char<? (string->list "abc")) \(\Rightarrow\) #t
(map char-upcase (string->list "abc")) \(\Rightarrow\) (#\A #\B #\C)

procedure: (list->string list)
returns: a string of the characters in list
libraries: (rnrs base), (rnrs)

list must consist entirely of characters.

list->string is the functional inverse of string->list. A program might use both procedures together, first converting a string into a list, then operating on this list to produce a new list, and finally converting the new list back into a string.
list->string may be defined without error checks as follows.

(define list->string
  (lambda (ls)
    (let ([s (make-string (length ls))])
      (do ([ls ls (cdr ls)] [i 0 (+ i 1)])
          ((null? ls) s)
          (string-set! s i (car ls))))))

(list->string '()) ⇒ ""
(list->string '(#\a #\b #\c)) ⇒ "abc"
(list->string
  (map char-upcase
    (string->list "abc"))) ⇒ "ABC"

Section 6.9. Vectors

Vectors are more convenient and efficient than lists for some applications. Whereas accessing an arbitrary element in a list requires a linear traversal of the list up to the selected element, arbitrary vector elements are accessed in constant time. The length of a vector is the number of elements it contains. Vectors are indexed by exact nonnegative integers, and the index of the first element of any vector is 0. The highest valid index for a given vector is one less than its length.

As with lists, the elements of a vector can be of any type, and a single vector can hold more than one type of object.

A vector is written as a sequence of objects separated by whitespace, preceded by the prefix #( and followed by ). For example, a vector consisting of the elements a, b, and c would be written #(a b c).

procedure: (vector obj ...)
returns: a vector of the objects obj ...
libraries: (rnrs base), (rnrs)

(vector) ⇒ #()
(vector 'a 'b 'c) ⇒ #(a b c)

procedure: (make-vector n)
procedure: (make-vector n obj)
returns: a vector of length n
libraries: (rnrs base), (rnrs)

n must be an exact nonnegative integer. If obj is supplied, each element of the vector is filled with obj; otherwise, the elements are unspecified.

(make-vector 0) ⇒ #()
(make-vector 0 '#(a)) ⇒ #()
(make-vector 5 '#(a)) ⇒ #(#(a) #(a) #(a) #(a) #(a))

procedure: (vector-length vector)
returns: the number of elements in vector
libraries: (rnrs base), (rnrs)

The length of a vector is always an exact nonnegative integer.
procedure: (vector-ref vector n)
returns: the n\textsuperscript{th} element (zero-based) of vector
libraries: (rnrs base), (rnrs)

n must be an exact nonnegative integer less than the length of vector.

(vector-ref '#(a b c) 0) ⇒ a
(vector-ref '#(a b c) 1) ⇒ b
(vector-ref '#(x y z w) 3) ⇒ w

procedure: (vector-set! vector n obj)
returns: unspecified
libraries: (rnrs base), (rnrs)

n must be an exact nonnegative integer less than the length of vector. vector-set! changes the n\textsuperscript{th} element of vector to obj.

(let ([v (vector 'a 'b 'c 'd 'e)])
    (vector-set! v 2 'x)
    v) ⇒ #(a b x d e)

procedure: (vector-fill! vector obj)
returns: unspecified
libraries: (rnrs base), (rnrs)

vector-fill! replaces each element of vector with obj. It may be defined without error checks as follows.

(define vector-fill!
  (lambda (v x)
    (let ([(n (vector-length v))]
        (do ([i 0 (+ i 1)])
            ((= i n))
              (vector-set! v i x))))))

(let ([v (vector 1 2 3)])
    (vector-fill! v 0)
    v) ⇒ #(0 0 0)

procedure: (vector->list vector)
returns: a list of the elements of vector
libraries: (rnrs base), (rnrs)

vector->list provides a convenient method for applying list-processing operations to vectors. It may be defined without error checks as follows.

(define vector->list
  (lambda (s)
    (do ([i (- (vector-length s) 1) (- i 1)])
        ))
(let ((v '#(1 2 3 4 5)))
  (apply * (vector->list v))) ⇒ 120

procedure: (list->vector list)
returns: a vector of the elements of list
libraries: (rnrs base), (rnrs)

list->vector is the functional inverse of vector->list. The two procedures are often used in combination to take advantage of a list-processing operation. A vector may be converted to a list with vector->list, this list processed in some manner to produce a new list, and the new list converted back into a vector with list->vector.

list->vector may be defined without error checks as follows.

(define list->vector
  (lambda (ls)
    (let ([s (make-vector (length ls))])
      (do ([ls ls (cdr ls)] [i 0 (+ i 1)])
        ((null? ls) s)
        (vector-set! s i (car ls)))))))

(list->vector '()) ⇒ #()
(list->vector '(a b c)) ⇒ #(a b c)

(let ((v '#(1 2 3 4 5)))
  (let ((ls (vector->list v)))
    (list->vector (map * 1s 1s)))) ⇒ #(1 4 9 16 25)

procedure: (vector-sort predicate vector)
returns: a vector containing the elements of vector, sorted according to predicate
procedure: (vector-sort! predicate vector)
returns: unspecified
libraries: (rnrs sorting), (rnrs)

predicate should be a procedure that expects two arguments and returns #t if its first argument must precede its second in the sorted vector. That is, if predicate is applied to two elements x and y, where x appears after y in the input vector, the predicate should return true only if x should appear before y in the output vector. If this constraint is met, vector-sort performs a stable sort, i.e., two elements are reordered only when necessary according to predicate. vector-sort! performs the sort destructively and does not necessarily perform a stable sort. Duplicate elements are not removed. predicate should not have any side effects.

vector-sort may call predicate up to n log n times, where n is the length of vector, while vector-sort! may call the predicate up to n^2 times. The looser bound for vector-sort! allows an implementation to use a quicksort algorithm, which may be faster in some cases than algorithms that have the tighter n log n bound.

(vector-sort < '#(3 4 2 1 2 5)) ⇒ #(1 2 2 3 4 5)
Section 6.10. Bytevectors

Bytevectors are vectors of raw binary data. Although nominally organized as a sequence of exact unsigned 8-bit integers, a bytevector can be interpreted as a sequence of exact signed 8-bit integers, exact signed or unsigned 16-bit, 32-bit, 64-bit, or arbitrary-precision integers, IEEE single or double floating-point numbers, or arbitrary combinations of the above.

The length of a bytevector is the number of 8-bit bytes it stores, and indices into a bytevector are always given as byte offsets. Any data element may be aligned at any byte offset, regardless of the underlying hardware’s alignment requirements, and may be represented using a specified endianness (see below) that differs from that prescribed by the hardware. Special, typically more efficient operators are provided for 16-, 32-, and 64-bit integers and single and double floats that are in their native format, i.e., with the endianness of the underlying hardware and stored at an index that is a multiple of the size in bytes of the integer or float.

The endianness of a multi-byte data value determines how it is laid out in memory. In big-endian format, the value is laid out with the more significant bytes at lower indices, while in little-endian format, the value is laid out with the more significant bytes at higher indices. When a bytevector procedure accepts an endianness argument, the argument may be the symbol big, representing the big-endian format, or the symbol little, representing the little-endian format. Implementations may extend these procedures to accept other endianness symbols. The native endianness of the implementation may be obtained via the procedure native-endianness.

Bytevectors are written with the #vu8 prefix in place of the # prefix for vectors, e.g., #vu8(1 2 3). The elements of a bytevector specified in this manner are always given as 8-bit unsigned exact integers, i.e., integers from 0 to 255 inclusive, written using any valid syntax for such numbers. Like strings, bytevectors are self-evaluating, so they need not be quoted.

'#vu8(1 2 3) ⇒ #vu8(1 2 3)
#vu8(1 2 3) ⇒ #vu8(1 2 3)
#vu8(#x3f #x7f #xbf #xff) ⇒ #vu8(63 127 191 255)

syntax: (endianness symbol)
returns: symbol
libraries: (rnrs bytevectors), (rnrs)

symbol must be the symbol little, the symbol big, or some other symbol recognized by the implementation as an endianness symbol. It is a syntax violation if symbol is not a symbol or if it is not recognized by the implementation as an endianness symbol.

(endianness little) ⇒ little
(endianness big) ⇒ big
(endianness "spam") ⇒ exception

procedure: (native-endianness)
returns: a symbol naming the implementation’s native endianness
libraries: (rnrs bytevectors), (rnrs)
The return value is the symbol little, the symbol big, or some other endianness symbol recognized by the implementation. It typically reflects the endianness of the underlying hardware.

(symbol? (native-endianness)) ⇒ #t

procedure: (make-bytevector n)
returns: a new bytevector of length n
libraries: (rnrs bytevectors), (rnrs)

If fill is supplied, each element of the bytevector is initialized to fill; otherwise, the elements are unspecified. The fill value must be a signed or unsigned 8-bit value, i.e., a value in the range -128 to 255 inclusive. A negative fill value is treated as its two's complement equivalent.

(make-bytevector 0) ⇒ #vu8()
(make-bytevector 0 7) ⇒ #vu8()
(make-bytevector 5 7) ⇒ #vu8(7 7 7 7 7)
(make-bytevector 5 -7) ⇒ #vu8(249 249 249 249 249)

procedure: (bytevector-length bytevector)
returns: the length of bytevector in 8-bit bytes
libraries: (rnrs bytevectors), (rnrs)

(bytevector-length #vu8()) ⇒ 0
(bytevector-length #vu8(1 2 3)) ⇒ 3
(bytevector-length (make-bytevector 300)) ⇒ 300

procedure: (bytevector=? bytevector₁ bytevector₂)
returns: #t if the relation holds, #f otherwise
libraries: (rnrs bytevectors), (rnrs)

Two bytevectors are equal by bytevector=? if and only if they have the same length and same contents.

(bytevector=? #vu8() #vu8()) ⇒ #t
(bytevector=? (make-bytevector 3 0) #vu8(0 0 0)) ⇒ #t
(bytevector=? (make-bytevector 5 0) #vu8(0 0 0)) ⇒ #f
(bytevector=? #vu8(1 127 128 255) #vu8(255 128 127 1)) ⇒ #f

procedure: (bytevector-fill! bytevector fill)
returns: unspecified
libraries: (rnrs bytevectors), (rnrs)

The fill value must be a signed or unsigned 8-bit value, i.e., a value in the range -128 to 255 inclusive. A negative fill value is treated as its two's complement equivalent.

bytevector-fill! replaces each element of bytevector with fill.

(let ([v (make-bytevector 6)])
  (bytevector-fill! v 255)
  v) ⇒ #vu8(255 255 255 255 255 255)

(let ([v (make-bytevector 6)])
  (bytevector-fill! v -128)
  v) ⇒ #vu8(128 128 128 128 128 128)
procedure: (bytevector-copy bytevector)
returns: a new bytevector that is a copy of bytevector
libraries: (rnrs bytevectors), (rnrs)

bytevector-copy creates a new bytevector with the same length and contents as bytevector.

(bytevector-copy #vu8(1 127 128 255)) ⇒ #vu8(1 127 128 255)

(let ([v #vu8(1 127 128 255)])
  (eq? v (bytevector-copy v))) ⇒ #f

procedure: (bytevector-copy! src src-start dst dst-start n)
returns: unspecified
libraries: (rnrs bytevectors), (rnrs)

src and dst must be bytevectors. src-start, dst-start, and n must be exact nonnegative integers. The sum of src-start and n must not exceed the length of src, and the sum of dst-start and n must not exceed the length of dst.

bytevector-copy! overwrites the n bytes of dst starting at dst-start with the n bytes of dst starting at src-start. This works even if dst is the same bytevector as src and the source and destination locations overlap. That is, the destination is filled with the bytes that appeared at the source before the operation began.

(define v1 #vu8(31 63 95 127 159 191 223 255))
(define v2 (make-bytevector 10 0))

(bytevector-copy! v1 2 v2 1 4)
v2 ⇒ #vu8(0 95 127 159 191 0 0 0 0 0)

<bytevector-copy! v1 5 v2 7 3>
v2 ⇒ #vu8(0 95 127 159 191 0 0 191 223 255)

<bytevector-copy! v2 3 v2 0 6>
v2 ⇒ #vu8(159 191 0 0 191 223 0 191 223 255)

<bytevector-copy! v2 0 v2 1 9>
v2 ⇒ #vu8(159 159 191 0 0 191 223 0 191 223)

procedure: (bytevector-u8-ref bytevector n)
returns: the 8-bit unsigned byte at index n (zero-based) of bytevector
libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer less than the length of bytevector.

The value is returned as an exact 8-bit unsigned integer, i.e., a value in the range 0 to 255 inclusive.

(bytevector-u8-ref #vu8(1 127 128 255) 0) ⇒ 1
(bytevector-u8-ref #vu8(1 127 128 255) 2) ⇒ 128
(bytevector-u8-ref #vu8(1 127 128 255) 3) ⇒ 255

procedure: (bytevector-s8-ref bytevector n)
returns: the 8-bit signed byte at index n (zero-based) of bytevector
libraries: (rnrs bytevectors), (rnrs)
\( n \) must be an exact nonnegative integer less than the length of \( \text{bytevector} \).

The value returned is an exact 8-bit signed integer, i.e., a value in the range -128 to 127 inclusive, and is the equivalent of the stored value treated as a two's complement value.

\[
\begin{align*}
(\text{bytevector-s8-ref} \ #\text{vu8}(1 127 128 255) 0) & \Rightarrow 1 \\
(\text{bytevector-s8-ref} \ #\text{vu8}(1 127 128 255) 1) & \Rightarrow 127 \\
(\text{bytevector-s8-ref} \ #\text{vu8}(1 127 128 255) 2) & \Rightarrow -128 \\
(\text{bytevector-s8-ref} \ #\text{vu8}(1 127 128 255) 3) & \Rightarrow -1
\end{align*}
\]

**procedure:** \( (\text{bytevector-u8-set!} \ \text{bytevector} \ n \ u8) \)

**returns:** unspecified

**libraries:** \( \text{rnrs bytevectors} \), \( \text{rnrs} \)

\( n \) must be an exact nonnegative integer less than the length of \( \text{bytevector} \). \( u8 \) must be an 8-bit unsigned value, i.e., a value in the range 0 to 255 inclusive.

\( \text{bytevector-u8-set!} \) changes the 8-bit value at index \( n \) (zero-based) of \( \text{bytevector} \) to \( u8 \).

\[
\begin{align*}
(\text{let} ([v (\text{make-bytevector} \ 5 \ -1)]) \\
(\text{bytevector-u8-set!} \ v \ 2 \ 128) \\
v) & \Rightarrow \#\text{vu8}(255 \ 255 \ 128 \ 255 \ 255)
\end{align*}
\]

**procedure:** \( (\text{bytevector-s8-set!} \ \text{bytevector} \ n \ s8) \)

**returns:** unspecified

**libraries:** \( \text{rnrs bytevectors} \), \( \text{rnrs} \)

\( n \) must be an exact nonnegative integer less than the length of \( \text{bytevector} \). \( s8 \) must be an 8-bit signed value, i.e., a value in the range -128 to 127 inclusive.

\( \text{bytevector-s8-set!} \) changes the 8-bit value at index \( n \) (zero-based) of \( \text{bytevector} \) to the two's complement equivalent of \( s8 \).

\[
\begin{align*}
(\text{let} ([v (\text{make-bytevector} \ 4 \ 0)]) \\
(\text{bytevector-s8-set!} \ v \ 1 \ 100) \\
(\text{bytevector-s8-set!} \ v \ 2 \ -100) \\
v) & \Rightarrow \#\text{vu8}(0 \ 100 \ 156 \ 0)
\end{align*}
\]

**procedure:** \( (\text{bytevector->u8-list} \ \text{bytevector}) \)

**returns:** a list of the 8-bit unsigned elements of \( \text{bytevector} \)

**libraries:** \( \text{rnrs bytevectors} \), \( \text{rnrs} \)

\[
\begin{align*}
(\text{bytevector->u8-list} \ (\text{make-bytevector} \ 0)) & \Rightarrow () \\
(\text{bytevector->u8-list} \ \#\text{vu8}(1 \ 127 \ 128 \ 255)) & \Rightarrow (1 \ 127 \ 128 \ 255)
\end{align*}
\]

\[
(\text{let} ([v \ #\text{vu8}(1 \ 2 \ 3 \ 255)]) \\
(\text{apply} \ * \ (\text{bytevector->u8-list} \ v))) & \Rightarrow 1530
\]

**procedure:** \( (\text{u8-list->bytevector} \ \text{list}) \)

**returns:** a new bytevector of the elements of \( \text{list} \)

**libraries:** \( \text{rnrs bytevectors} \), \( \text{rnrs} \)

\( \text{list} \) must consist entirely of exact 8-bit unsigned integers, i.e., values in the range 0 to 255 inclusive.
(u8-list->bytevector '()) ⇒ #vu8()
(u8-list->bytevector '(1 127 128 255)) ⇒ #vu8(1 127 128 255)

(let ([v #vu8(1 2 3 4 5)])
  (let ([ls (bytevector->u8-list v)])
    (u8-list->bytevector (map * ls ls))) ⇒ #vu8(1 4 9 16 25)

procedure: (bytevector-u16-native-ref bytevector n)
returns: the 16-bit unsigned integer at index n (zero-based) of bytevector

procedure: (bytevector-s16-native-ref bytevector n)
returns: the 16-bit signed integer at index n (zero-based) of bytevector

procedure: (bytevector-u32-native-ref bytevector n)
returns: the 32-bit unsigned integer at index n (zero-based) of bytevector

procedure: (bytevector-s32-native-ref bytevector n)
returns: the 32-bit signed integer at index n (zero-based) of bytevector

procedure: (bytevector-u64-native-ref bytevector n)
returns: the 64-bit unsigned integer at index n (zero-based) of bytevector

procedure: (bytevector-s64-native-ref bytevector n)
returns: the 64-bit signed integer at index n (zero-based) of bytevector

libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer. It indexes the starting byte of the value and must be a multiple of the number of bytes occupied by the value: 2 for 16-bit values, 4 for 32-bit values, and 8 for 64-bit values. The sum of n and the number of bytes occupied by the value must not exceed the length of bytevector. The native endianness is assumed.

The return value is an exact integer in the appropriate range for the number of bytes occupied by the value. Signed values are the equivalent of the stored value treated as a two's complement value.

(define v #vu8(#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98))

If native endianness is big:

(bytevector-u16-native-ref v 2) ⇒ #xfe56
(bytevector-s16-native-ref v 2) ⇒ #x-1aa
(bytevector-s16-native-ref v 6) ⇒ #x789b

(bytevector-u32-native-ref v 0) ⇒ #x1234fe56
(bytevector-s32-native-ref v 0) ⇒ #x1234fe56
(bytevector-s32-native-ref v 4) ⇒ #x-23458768

(bytevector-u64-native-ref v 0) ⇒ #x1234fe56dcba7898
(bytevector-s64-native-ref v 0) ⇒ #x1234fe56dcba7898

If native endianness is little:

(bytevector-u16-native-ref v 2) ⇒ #x56fe
(bytevector-s16-native-ref v 2) ⇒ #x56fe
(bytevector-s16-native-ref v 6) ⇒ #x-6788

(bytevector-u32-native-ref v 0) ⇒ #x56fe3412
(bytevector-s32-native-ref v 0) ⇒ #x56fe3412
(bytevector-s32-native-ref v 4) ⇒ #x-67874524
procedure: (bytevector-u16-native-set! bytevector n u16)
procedure: (bytevector-s16-native-set! bytevector n s16)
procedure: (bytevector-u32-native-set! bytevector n u32)
procedure: (bytevector-s32-native-set! bytevector n s32)
procedure: (bytevector-u64-native-set! bytevector n u64)
procedure: (bytevector-s64-native-set! bytevector n s64)

returns: unspecified

libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer. It indexes the starting byte of the value and must be a multiple of the number of bytes occupied by the value: 2 for 16-bit values, 4 for 32-bit values, and 8 for 64-bit values. The sum of n and the number of bytes occupied by the value must not exceed the length bytevector. u16 must be a 16-bit unsigned value, i.e., a value in the range 0 to 2^{16} - 1 inclusive; s16 must be a 16-bit signed value, i.e., a value in the range -2^{15} to 2^{15} - 1 inclusive; u32 must be a 32-bit unsigned value, i.e., a value in the range 0 to 2^{32} - 1 inclusive; s32 must be a 32-bit signed value, i.e., a value in the range -2^{31} to 2^{31} - 1 inclusive; u64 must be a 64-bit unsigned value, i.e., a value in the range 0 to 2^{64} - 1 inclusive; and s64 must be a 64-bit signed value, i.e., a value in the range -2^{63} to 2^{63} - 1 inclusive. The native endianness is assumed.

These procedures store the given value in the 2, 4, or 8 bytes starting at index n (zero-based) of bytevector. Negative values are stored as their two's complement equivalent.

(define v (make-bytevector 8 0))
(bytevector-u16-native-set! v 0 #xfe56)
(bytevector-s16-native-set! v 2 #x-1aa)
(bytevector-s16-native-set! v 4 #x7898)

If native endianness is big:

v ⇒ #vu8(#xfe #x56 #x56 #x78 #x98 #x00 #x00)

If native endianness is little:

v ⇒ #vu8(#x56 #xfe #x56 #x98 #x78 #x00 #x00)

(define v (make-bytevector 16 0))
(bytevector-u32-native-set! v 0 #x1234fe56)
(bytevector-s32-native-set! v 4 #x1234fe56)
(bytevector-s32-native-set! v 8 #x-23458768)

If native endianness is big:

v ⇒ #vu8(#x12 #x34 #xfe #x56 #x12 #x34 #xfe #x56
    #xdc #xba #x78 #x98 #x00 #x00 #x00 #x00)

If native endianness is little:

v ⇒ #vu8(#x56 #xfe #x34 #x12 #x56 #xfe #x34 #x12
    #x98 #x78 #xba #xdc #x00 #x00 #x00 #x00)

(define v (make-bytevector 24 0))
(bytevector-u64-native-set! v 0 #x1234fe56dcba7898)
If native endianness is big:

```
(v 8 #x1234fe56dcba7898)
(v 16 #x-67874523a901cbee)
```

If native endianness is little:

```
(v #vu8(#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x98 #x78 #xba #xdc #x56 #xfe #x34 #x12)
```

```
procedure: (bytevector-u16-ref bytevector n eness)
returns: the 16-bit unsigned integer at index n (zero-based) of bytevector
```

```
procedure: (bytevector-s16-ref bytevector n eness)
returns: the 16-bit signed integer at index n (zero-based) of bytevector
```

```
procedure: (bytevector-u32-ref bytevector n eness)
returns: the 32-bit unsigned integer at index n (zero-based) of bytevector
```

```
procedure: (bytevector-s32-ref bytevector n eness)
returns: the 32-bit signed integer at index n (zero-based) of bytevector
```

```
procedure: (bytevector-u64-ref bytevector n eness)
returns: the 64-bit unsigned integer at index n (zero-based) of bytevector
```

```
procedure: (bytevector-s64-ref bytevector n eness)
returns: the 64-bit signed integer at index n (zero-based) of bytevector
```

```
libraries: (rnrs bytevectors), (rnrs)
```

If native endianness is big:

```
(v ⇒ #vu8(#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x98 #x78 #xba #xdc #x56 #xfe #x34 #x12)
```

If native endianness is little:

```
(v ⇒ #vu8(#x98 #x78 #xba #xdc #x56 #xfe #x34 #x12 #x98 #x78 #xba #xdc #x56 #xfe #x34 #x12 #x12 #x34 #xfe #x56 #xdc #xba #x78 #x98)
```

```
(n must be an exact nonnegative integer and indexes the starting byte of the value. The sum of n and the number of bytes occupied by the value (2 for 16-bit values, 4 for 32-bit values, and 8 for 32-bit values) must not exceed the length of bytevector. n need not be a multiple of the number of bytes occupied by the value. eness must be a valid endianness symbol naming the endianness.
```

The return value is an exact integer in the appropriate range for the number of bytes occupied by the value. Signed values are the equivalent of the stored value treated as a two's complement value.

```
(define v #vu8(#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x9a #x76))
(bytevector-u16-ref v 0 (endianness big)) ⇒ #x1234
(bytevector-s16-ref v 1 (endianness big)) ⇒ #x34fe
(bytevector-s16-ref v 5 (endianness big)) ⇒ #x-4588
```

```
(bytevector-u32-ref v 2 'big) ⇒ #x56dcba
(bytevector-s32-ref v 3 'big) ⇒ #x56dcba78
(bytevector-s32-ref v 4 'big) ⇒ #x-23458768
```

```
(bytevector-u64-ref v 0 'big) ⇒ #x1234fe56dcba7898
(bytevector-s64-ref v 1 'big) ⇒ #x34fe56dcba7898a
```

```
(bytevector-u16-ref v 0 (endianness little)) ⇒ #x3412
(bytevector-s16-ref v 1 (endianness little)) ⇒ #x-1cc
(bytevector-s16-ref v 5 (endianness little)) ⇒ #x78ba
```

```
(bytevector-u32-ref v 2 'little) ⇒ #xbadc56fe
```

Section 6.10. Bytevectors
\[(\text{bytevector-s32-ref v 3 'little}) \Rightarrow \#x78badc56\]
\[(\text{bytevector-s32-ref v 4 'little}) \Rightarrow \#x-67874524\]

\[(\text{bytevector-u64-ref v 0 'little}) \Rightarrow \#x9878badc56fe3412\]
\[(\text{bytevector-s64-ref v 1 'little}) \Rightarrow \#x-6567874523a901cc\]

**procedure**: \((\text{bytevector-u16-set! bytevector n u16 eness})\)
**procedure**: \((\text{bytevector-s16-set! bytevector n s16 eness})\)
**procedure**: \((\text{bytevector-u32-set! bytevector n u32 eness})\)
**procedure**: \((\text{bytevector-s32-set! bytevector n s32 eness})\)
**procedure**: \((\text{bytevector-u64-set! bytevector n u64 eness})\)
**procedure**: \((\text{bytevector-s64-set! bytevector n s64 eness})\)
**returns**: unspecified
**libraries**: (rnrs bytevectors), (rnrs)

\(n\) must be an exact nonnegative integer and indexes the starting byte of the value. The sum of \(n\) and the number of bytes occupied by the value must not exceed the length of \(\text{bytevector}\). \(n\) need not be a multiple of the number of bytes occupied by the value. \(u16\) must be a 16-bit unsigned value, i.e., a value in the range 0 to \(2^{16}-1\) inclusive; \(s16\) must be a 16-bit signed value, i.e., a value in the range \(-2^{15}\) to \(2^{15} - 1\) inclusive; \(u32\) must be a 32-bit unsigned value, i.e., a value in the range 0 to \(2^{32} - 1\) inclusive; \(s32\) must be a 32-bit signed value, i.e., a value in the range \(-2^{31}\) to \(2^{31} - 1\) inclusive; \(u64\) must be a 64-bit unsigned value, i.e., a value in the range 0 to \(2^{64} - 1\) inclusive; and \(s64\) must be a 64-bit signed value, i.e., a value in the range \(-2^{63}\) to \(2^{63} - 1\) inclusive. \(eness\) must be a valid endianness symbol naming the endianness.

These procedures store the given value in the 2, 4, or 8 bytes starting at index \(n\) (zero-based) of \(\text{bytevector}\). Negative values are stored as their two's complement equivalent.

\[
\begin{align*}
\text{(define v (make-bytevector 8 0))} \\
\text{(bytevector-u16-set! v 0 \#xfe56 (endianness big))} \\
\text{(bytevector-s16-set! v 3 \#x-1aa (endianness little))} \\
\text{(bytevector-s16-set! v 5 \#x7898 (endianness big))} \\
\text{v} & \Rightarrow \#vu8(\#xfe \#x56 \#x0 \#x56 \#xfe \#x78 \#x98 \#x0) \\
\end{align*}
\]

\[
\begin{align*}
\text{(define v (make-bytevector 16 0))} \\
\text{(bytevector-u32-set! v 0 \#x1234fe56 'little)} \\
\text{(bytevector-s32-set! v 6 \#x1234fe56 'big)} \\
\text{(bytevector-s32-set! v 11 \#x-23458768 'little)} \\
\text{v} & \Rightarrow \#vu8(\#x56 \#xfe \#x34 \#x12 \#x0 \#x0 \\
\text{\qquad \#x12 \#x34 \#xfe \#x56 \#x0} \\
\text{\qquad \#x98 \#x78 \#xBA \#xdc \#x0) \\
\end{align*}
\]

\[
\begin{align*}
\text{(define v (make-bytevector 28 0))} \\
\text{(bytevector-u64-set! v 0 \#x1234fe56dcb87898 'little)} \\
\text{(bytevector-s64-set! v 10 \#x1234fe56dcb87898 'big)} \\
\text{(bytevector-s64-set! v 19 \#x-67874523a901cbee 'big)} \\
\text{v} & \Rightarrow \#vu8(\#x98 \#x78 \#xBA \#xdc \#x56 \#xfe \#x34 \#x12 \#x0 \#x0 \\
\text{\qquad \#x12 \#x34 \#xfe \#x56 \#xdc \#xBA \#x78 \#x98 \#x0} \\
\text{\qquad \#x98 \#x78 \#xBA \#xdc \#x56 \#xfe \#x34 \#x12 \#x0) \\
\end{align*}
\]

**procedure**: \((\text{bytevector-uint-ref bytevector n eness size})\)
**returns**: the \(size\)-byte unsigned integer at index \(n\) (zero-based) of \(\text{bytevector}\)
**procedure**: \((\text{bytevector-sint-ref bytevector n eness size})\)
**returns**: the \(size\)-byte signed integer at index \(n\) (zero-based) of \(\text{bytevector}\)
**libraries**: (rnrs bytevectors), (rnrs)
\( n \) must be an exact nonnegative integer and indexes the starting byte of the value. \( \text{size} \) must be an exact positive integer and specifies the number of bytes occupied by the value. The sum of \( n \) and \( \text{size} \) must not exceed the length of \texttt{bytevector}. \( n \) need not be a multiple of the number of bytes occupied by the value. \( \text{eness} \) must be a valid endianness symbol naming the endianness.

The return value is an exact integer in the appropriate range for the number of bytes occupied by the value. Signed values are the equivalent of the stored value treated as a two’s complement value.

\[
\begin{align*}
(\text{define } v & \textbf{#vu8}((\text{x12 } \text{ #x34 } \text{ #xfe } \text{ #x56 } \text{ #xdc } \text{ #xba } \text{ #x78 } \text{ #x98 } \text{ #x9a } \text{ #x76})) \\
(\text{bytevector-uint-ref } v & 0 \textbf{ 'big 1}) \Rightarrow \text{#x12} \\
(\text{bytevector-uint-ref } v & 0 \textbf{ 'little 1}) \Rightarrow \text{#x12} \\
(\text{bytevector-uint-ref } v & 1 \textbf{ 'big 3}) \Rightarrow \text{#x34fe56} \\
(\text{bytevector-uint-ref } v & 2 \textbf{ 'little 7}) \Rightarrow \text{#x9a987badc56fe} \\
(\text{bytevector-sint-ref } v & 2 \textbf{ 'big 1}) \Rightarrow \text{#-02} \\
(\text{bytevector-sint-ref } v & 1 \textbf{ 'little 6}) \Rightarrow \text{#x78badc56fe34} \\
(\text{bytevector-sint-ref } v & 2 \textbf{ 'little 7}) \Rightarrow \text{#-6567874523a902} \\
(\text{bytevector-sint-ref } (\text{make-bytevector 1000 } -1) & 0 \textbf{ 'big 1000}) \Rightarrow \text{-1} \\
\end{align*}
\]

\textbf{procedure:} (\text{bytevector-uint-set!} \text{ bytevector } n \text{ uint eness size})
\textbf{procedure:} (\text{bytevector-sint-set!} \text{ bytevector } n \text{ sint eness size})
\textbf{returns:} unspecified
\textbf{libraries:} (rnrs bytevectors), (rnrs)

\( n \) must be an exact nonnegative integer and indexes the starting byte of the value. \( \text{size} \) must be an exact positive integer and specifies the number of bytes occupied by the value. The sum of \( n \) and \( \text{size} \) must not exceed the length of \texttt{bytevector}. \( n \) need not be a multiple of the number of bytes occupied by the value. \( \text{uint} \) must be an exact integer in the range \( 0 \) to \( 2^{\text{size} \cdot 8} - 1 \) inclusive. \( \text{sint} \) must be an exact integer in the range \(-2^{\text{size} \cdot 8 - 1} \) to \( 2^{\text{size} \cdot 8 - 1} - 1 \) inclusive. \( \text{eness} \) must be a valid endianness symbol naming the endianness.

These procedures store the given value in the \( \text{size} \) bytes starting at index \( n \) (zero-based) of \texttt{bytevector}. Negative values are stored as their two’s complement equivalent.

\[
\begin{align*}
(\text{define } v & (\text{make-bytevector 5 0})) \\
(\text{bytevector-uint-set! } v & 1 \textbf{ #x123456 (endianness big) 3}) \\
v & \Rightarrow \text{#vu8}(0 \textbf{ #x12 } \text{ #x34 } \text{ #x56 } \text{ 0}) \\
(\text{define } v & (\text{make-bytevector 7 } -1)) \\
(\text{bytevector-sint-set! } v & 1 \textbf{ #x-8000000000 (endianness little) 5}) \\
v & \Rightarrow \text{#vu8}(\text{#xff 0 0 0 0 #x80 #xff}) \\
\end{align*}
\]

\textbf{procedure:} (\text{bytevector->uint-list} \text{ bytevector eness size})
\textbf{returns:} a new list of the \( \text{size} \)-bit unsigned elements of \texttt{bytevector}
\textbf{procedure:} (\text{bytevector->sint-list} \text{ bytevector eness size})
\textbf{returns:} a new list of the \( \text{size} \)-bit signed elements of \texttt{bytevector}
\textbf{libraries:} (rnrs bytevectors), (rnrs)

\( \text{eness} \) must be a valid endianness symbol naming the endianness. \( \text{size} \) must be an exact positive integer and specifies the number of bytes occupied by the value. It must be a value that evenly divides the length of \texttt{bytevector}.

\[
(\text{bytevector->uint-list } (\text{make-bytevector 0} \textbf{ 'little 3}) \Rightarrow \text{()} \\
\]

\textbf{Section 6.10. Bytevectors}
(let ([v #vu8(1 2 3 4 5 6)])
  (bytevector->uint-list v 'big 3)) => (#x010203 #x040506)

(let ([v (make-bytevector 80 -1)])
  (bytevector->sint-list v 'big 20)) => (-1 -1 -1 -1)

procedure: (uint-list->bytevector list eness size)
procedure: (sint-list->bytevector list eness size)
returns: a new bytevector of the elements of list
libraries: (rnrs bytevectors), (rnrs)

eness must be a valid endianness symbol naming the endianness. size must be an exact positive integer and specifies the number of bytes occupied by the value. For uint-list->bytevector, list must consist entirely of size-byte exact unsigned integers, i.e., values in the range 0 to \(2^{size \cdot 8} - 1\) inclusive. For sint-list->bytevector, list must consist entirely of size-byte exact signed integers, i.e., values in the range \(-2^{size \cdot 8} - 1\) to \(2^{size \cdot 8} - 1\) inclusive. Each value occupies size bytes in the resulting bytevector, whose length is thus size times the length of list.

(uint-list->bytevector '() 'big 25) => #vu8()
(sint-list->bytevector '(0 -1) 'big 3) => #vu8(0 0 0 #xff #xff #xff)

(define (f size)
  (let ([ls (list (- (expt 2 (- (* 8 size) 1)))
               (- (expt 2 (- (* 8 size) 1))) 1)])
    (sint-list->bytevector ls 'little size)))
(f 6) => #vu8(#x00 #x00 #x00 #x00 #x00 #x80 #xff #xff #xff #xff #xff #x7f)

procedure: (bytevector-ieee-single-native-ref bytevector n)
returns: the single floating-point value at index n (zero-based) of bytevector
procedure: (bytevector-ieee-double-native-ref bytevector n)
returns: the double floating-point value at index n (zero-based) of bytevector
libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer. It indexes the starting byte of the value and must be a multiple of the number of bytes occupied by the value: 4 for single floats, 8 for double. The sum of n and the number of bytes occupied by the value must not exceed the length of bytevector. The native endianness is assumed.

The return value is an inexact real number. Examples appear after the mutation operators below.

procedure: (bytevector-ieee-single-native-set! bytevector n x)
procedure: (bytevector-ieee-double-native-set! bytevector n x)
returns: unspecified
libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer. It indexes the starting byte of the value and must be a multiple of the number of bytes occupied by the value: 4 for single floats, 8 for double. The sum of n and the number of bytes occupied by the value must not exceed the length of bytevector. The native endianness is assumed.

These procedures store the given value as an IEEE-754 single or double floating-point value at index n (zero-based) of bytevector.

(define v (make-bytevector 8 0))
(bytevector-ieee-single-native-set! v 0 .125)
(bytevector-ieee-single-native-set! v 4 -3/2)
(list
  (bytevector-ieee-single-native-ref v 0)
  (bytevector-ieee-single-native-ref v 4)) ⇒ (0.125 -1.5)

(bytevector-ieee-double-native-set! v 0 1e23)
(bytevector-ieee-double-native-ref v 0) ⇒ 1e23

procedure: (bytevector-ieee-single-ref bytevector n eness)
returns: the single floating-point value at index n (zero-based) of bytevector

procedure: (bytevector-ieee-double-ref bytevector n eness)
returns: the double floating-point value at index n (zero-based) of bytevector

libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer and indexes the starting byte of the value. The sum of n and the number of bytes occupied by the value (4 for a single float, 8 for a double) must not exceed the length of bytevector. n need not be a multiple of the number of bytes occupied by the value. eness must be a valid endianness symbol naming the endianness.

The return value is an inexact real number. Examples appear after the mutation operators below.

procedure: (bytevector-ieee-single-set! bytevector n x eness)
procedure: (bytevector-ieee-double-set! bytevector n x eness)
returns: unspecified

libraries: (rnrs bytevectors), (rnrs)

n must be an exact nonnegative integer and indexes the starting byte of the value. The sum of n and the number of bytes occupied by the value (4 for a single float, 8 for a double) must not exceed the length of bytevector. n need not be a multiple of the number of bytes occupied by the value. eness must be a valid endianness symbol naming the endianness.

These procedures store the given value as an IEEE-754 single or double floating-point value at index n (zero-based) of bytevector.

(define v (make-bytevector 10 #xc7))
(bytevector-ieee-single-set! v 1 .125 'little)
(bytevector-ieee-single-set! v 6 -3/2 'big)
(list
  (bytevector-ieee-single-ref v 1 'little)
  (bytevector-ieee-single-ref v 6 'big)) ⇒ (0.125 -1.5)

v ⇒ #vu8(#xc7 #x0 #x0 #x3e #xc7 #xbf #xc0 #x0 #x0)

(bytevector-ieee-double-set! v 1 1e23 'big)
(bytevector-ieee-double-ref v 1 'big) ⇒ 1e23

Section 6.11. Symbols

Symbols are used for a variety of purposes as symbolic names in Scheme programs. Strings could be used for most of the same purposes, but an important characteristic of symbols makes comparisons between symbols much more efficient. This characteristic is that two symbols with the same name are identical in the sense of eq?. The reason is that the Scheme reader (invoked by get-datum and read) and the procedure string->symbol catalog symbols in an internal symbol table and always return the same symbol whenever the same name is encountered. Thus, no character-by-character comparison is needed, as would be
needed to compare two strings.

The property that two symbols may be compared quickly for equivalence makes them ideally suited for use as identifiers in the representation of programs, allowing fast comparison of identifiers. This property also makes symbols useful for a variety of other purposes. For example, symbols might be used as messages passed between procedures, labels for list-structured records, or names for objects stored in an association list (see \texttt{assq} in Section 6.3).

Symbols are written without double quotes or other bracketing characters. Parentheses, double quotes, spaces, and most other characters with a special meaning to the Scheme reader are not allowed within the printed representation of a symbol. These and any other Unicode character may appear anywhere within the printed representation of a symbol with the syntax \texttt{\#\char{xn};}, where \texttt{n} consists of one or more hexadecimal digits and represents a valid Unicode scalar value.

The grammar for symbols on page 458 gives a precise definition of the syntax of symbols.

\begin{verbatim}
procedure: (symbol=? symbol1 symbol2)
returns: #t if the two symbols are the same, #f otherwise
libraries: (rnrs base), (rnrs)
\end{verbatim}

Symbols can also be compared with \texttt{eq?}, which is typically more efficient than \texttt{symbol=?}.

\begin{verbatim}
(symbol=? 'a 'a) ⇒ #t
(symbol=? 'a (string->symbol "a")) ⇒ #t
(symbol=? 'a 'b) ⇒ #f
\end{verbatim}

\begin{verbatim}
procedure: (string->symbol string)
returns: a symbol whose name is string
libraries: (rnrs base), (rnrs)
\end{verbatim}

\texttt{string->symbol} records all symbols it creates in an internal table that it shares with the system reader. If a symbol whose name is equivalent to \texttt{string} (according to the predicate \texttt{string=?}) already exists in the table, this symbol is returned. Otherwise, a new symbol is created with \texttt{string} as its name; this symbol is entered into the table and returned.

The effect of modifying a string after it is used as an argument to \texttt{string->symbol} is unspecified.

\begin{verbatim}
(string->symbol "x") ⇒ x
(eq? (string->symbol "x") 'x) ⇒ #t
(eq? (string->symbol "X") 'x) ⇒ #f
(eq? (string->symbol "x")
  (string->symbol "x")) ⇒ #t
(string->symbol "") ⇒ \x28;\x29;
\end{verbatim}

\begin{verbatim}
procedure: (symbol->string symbol)
returns: a string, the name of symbol
libraries: (rnrs base), (rnrs)
\end{verbatim}

The string returned by \texttt{symbol->string} should be treated as immutable. Unpredictable behavior can result if a string passed to \texttt{string->symbol} is altered with \texttt{string-set!} or by any other means.
Section 6.12. Booleans

While every Scheme object has a truth value when used in a conditional context, with every object but #f counting as true, Scheme provides the dedicated true value #t for use when a value of an expression should convey nothing more than that it is true.

procedure: (boolean=? boolean1 boolean2)
returns: #t if the two booleans are the same, #f otherwise
libraries: (rnrs base), (rnrs)

The boolean values #t and #f may also be compared with eq?, which is typically more efficient than boolean=?.

(boolean=? #t #t) ⇒ #t
(boolean=? #t #f) ⇒ #f
(boolean=? #t (< 3 4)) ⇒ #t

Section 6.13. Hashtables

Hashtables represent sets of associations between arbitrary Scheme values. They serve essentially the same purpose as association lists (see page 165) but are typically much faster when large numbers of associations are involved.

procedure: (make-eq-hashtable)
procedure: (make-eq-hashtable size)
returns: a new mutable eq hashtable
libraries: (rnrs hashtables), (rnrs)

If size is provided, it must be a nonnegative exact integer indicating approximately how many elements the hashtable should initially hold. Hashtables grow as needed, but when the hashtable grows it generally must rehash all of the existing elements. Providing a nonzero size can help limit the amount of rehashing that must be done as the table is initially populated.

An eq hashtable compares keys using the eq? (pointer equality) procedure and typically employs a hash function based on object addresses. Its hash and equivalence functions are suitable for any Scheme object.

(define ht1 (make-eq-hashtable))
(define ht2 (make-eq-hashtable 32))

procedure: (make-eqv-hashtable)
procedure: (make-eqv-hashtable size)
returns: a new mutable eqv hashtable
libraries: (rnrs hashtables), (rnrs)

If size is provided, it must be a nonnegative exact integer indicating approximately how many elements the hashtable should initially hold. Hashtables grow as needed, but when the hashtable grows it generally must rehash all of the existing elements. Providing a nonzero size can help limit the amount of rehashing that must be done as the table is initially populated.
An eqv hashtable compares keys using the \texttt{eqv?} procedure and typically employs a hash function based on object addresses for objects that are identifiable with \texttt{eq?}. Its hash and equivalence functions are suitable for any Scheme object.

\begin{verbatim}
procedure: (make-hashtable hash equiv?)
procedure: (make-hashtable hash equiv? size)
returns: a new mutable hashtable
libraries: (rnrs hashtables), (rnrs)
\end{verbatim}

\texttt{hash} and \texttt{equiv?} must be procedures. If \texttt{size} is provided, it must be a nonnegative exact integer indicating approximately how many elements the hashtable should initially hold. Hashtables grow as needed, but when the hashtable grows it generally must rehash all of the existing elements. Providing a nonzero \texttt{size} can help limit the amount of rehashing that must be done as the table is initially populated.

The new hashtable computes hash values using \texttt{hash} and compares keys using \texttt{equiv?}, neither of which should modify the hashtable. \texttt{equiv?} should compare two keys and return false only if the two keys should be distinguished. \texttt{hash} should accept a key as an argument and return a nonnegative exact integer value that is the same each time it is called with arguments that \texttt{equiv?} does not distinguish. The \texttt{hash} and \texttt{equiv?} procedures need not accept arbitrary inputs as long as the hashtable is used only for keys that they do accept, and both procedures may assume that the keys are immutable as long as the keys are not modified while they have associations stored in the table. The hashtable operation may call \texttt{hash} and \texttt{equiv?} once, not at all, or multiple times for each hashtable operation.

\begin{verbatim}
(define ht (make-hashtable string-hash string=?))
procedure: (hashtable-mutable? hashtable)
returns: \#t if hashtable is mutable, \#f otherwise
libraries: (rnrs hashtables), (rnrs)
\end{verbatim}

Hashtables returned by one of the hashtable creation procedures above are mutable, but those created by \texttt{hashtable-copy} may be immutable. Immutable hashtables cannot be altered by any of the procedures \texttt{hashtable-set!}, \texttt{hashtable-update!}, \texttt{hashtable-delete!}, or \texttt{hashtable-clear!}.

\begin{verbatim}
(hashtable-mutable? (make-eq-hashtable)) ⇒ \#t
(hashtable-mutable? (hashtable-copy (make-eq-hashtable))) ⇒ \#f
procedure: (hashtable-hash-function hashtable)
returns: the hash function associated with hashtable
procedure: (hashtable-equivalence-function hashtable)
returns: the equivalence function associated with hashtable
libraries: (rnrs hashtables), (rnrs)
\end{verbatim}

\texttt{hashtable-hash-function} returns \#f for eq and eqv hashtables.

\begin{verbatim}
(hashtable-hash-function ht) ⇒ \#f
(eq? (hashtable-equivalence-function ht) eq?) ⇒ \#t
\end{verbatim}

\begin{verbatim}
(define ht (make-eq-hashtable))
(hashtable-hash-function ht) ⇒ \#f
(eq? (hashtable-equivalence-function ht) string=?)) ⇒ \#t
\end{verbatim}

\begin{verbatim}
procedure: (equal-hash obj)
procedure: (string-hash string)
\end{verbatim}
procedure: (string-ci-hash string)
procedure: (symbol-hash symbol)
returns: an exact nonnegative integer hash value
libraries: (rnrs hashtables), (rnrs)

These procedures are hash functions suitable for use with the appropriate Scheme predicate: equal? for equal-hash, string=? for string-hash, string-ci=? for string-ci-hash, and symbol=? (or eq?) for symbol-hash. The hash values returned by equal-hash, string-hash, and string-ci-hash are typically dependent on the current structure and contents of the input values and are thus unsuitable if keys are modified while they have associations in a hashtable.

procedure: (hashtable-set! hashtable key obj)
returns: unspecified
libraries: (rnrs hashtables), (rnrs)

hashtable must be a mutable hashtable. key should be an appropriate key for the hashtable's hash and equivalence functions. obj may be any Scheme object.

hashtable-set! associates key with obj in hashtable, replacing the existing association, if any.

(define ht (make-eq-hashtable))
(hashtable-set! ht 'a 73)

procedure: (hashtable-ref hashtable key default)
returns: see below
libraries: (rnrs hashtables), (rnrs)

key should be an appropriate key for the hashtable's hash and equivalence functions. default may be any Scheme object.

hashtable-ref returns the value associated with key in hashtable. If no value is associated with key in hashtable, hashtable-ref returns default.

(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))

(define eqht (make-eq-hashtable))
(hashtable-set! eqht p1 73)
(hashtable-ref eqht p1 55) ⇒ 73
(hashtable-ref eqht p2 55) ⇒ 55

(define equalht (make-hashtable equal-hash equal?))
(hashtable-set! equalht p1 73)
(hashtable-ref equalht p1 55) ⇒ 73
(hashtable-ref equalht p2 55) ⇒ 73

procedure: (hashtable-contains? hashtable key)
returns: #t if an association for key exists in hashtable, #f otherwise
libraries: (rnrs hashtables), (rnrs)

key should be an appropriate key for the hashtable's hash and equivalence functions.

(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 73)
(hashtable-contains? ht p1) ⇒ #t
(hashtable-contains? ht p2) ⇒ #f

**procedure:** (hashtable-update! hashtable key procedure default)

**returns:** unspecified

**libraries:** (rnrs hashtables), (rnrs)

`hashtable` must be a mutable hashtable. `key` should be an appropriate key for the hashtable's hash and equivalence functions. `default` may be any Scheme object. `procedure` should accept one argument, should return one value, and should not modify `hashtable`.

`hashtable-update!` applies `procedure` to the value associated with `key` in `hashtable`, or to `default` if no value is associated with `key` in `hashtable`. If `procedure` returns, `hashtable-update!` associates `key` with the value returned by `procedure`, replacing the old association, if any.

A version of `hashtable-update!` that does not verify that it receives arguments of the proper type might be defined as follows.

```
(define hashtable-update!
  (lambda (ht key proc value)
    (hashtable-set! ht key
    (proc (hashtable-ref ht key value)))))
```

An implementation may, however, be able to implement `hashtable-update!` more efficiently by avoiding multiple hash computations and hashtable lookups.

```
(define ht (make-eq-hashtable))
(hashtable-update! ht 'a
  (lambda (x) (* x 2)) 55)
(hashtable-ref ht 'a 0) ⇒ 110
(hashtable-update! ht 'a
  (lambda (x) (* x 2))
  0)
(hashtable-ref ht 'a 0) ⇒ 220
```

**procedure:** (hashtable-delete! hashtable key)

**returns:** unspecified

**libraries:** (rnrs hashtables), (rnrs)

`hashtable` must be a mutable hashtable. `key` should be an appropriate key for the hashtable's hash and equivalence functions.

`hashtable-delete!` drops any association for `key` from `hashtable`.

```
(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 73)
(hashtable-contains? ht p1) ⇒ #t
(hashtable-delete! ht p1)
```
(hashtable-contains? ht p1) ⇒ #f
(hashtable-contains? ht p2) ⇒ #f
(hashtable-delete! ht p2)

procedure: (hashtable-size hashtable)
returns: number of entries in hashtable
libraries: (rnrs hashtables), (rnrs)

(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-size ht) ⇒ 0
(hashtable-set! ht p1 73)
(hashtable-size ht) ⇒ 1
(hashtable-delete! ht p1)
(hashtable-size ht) ⇒ 0

procedure: (hashtable-copy hashtable)
procedure: (hashtable-copy hashtable mutable?)
returns: a new hashtable containing the same entries as hashtable
libraries: (rnrs hashtables), (rnrs)

If mutable? is present and not false, the copy is mutable; otherwise, the copy is immutable.

(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(hashtable-set! ht p1 "c")
(define ht-copy (hashtable-copy ht))
(hashtable-mutable? ht-copy) ⇒ #f
(hashtable-delete! ht p1)
(hashtable-delete! ht-copy p1) ⇒ #f
(hashtable-ref ht p1 #f) ⇒ #f
(hashtable-ref ht-copy p1 #f) ⇒ "c"

procedure: (hashtable-clear! hashtable)
procedure: (hashtable-clear! hashtable size)
returns: unspecified
libraries: (rnrs hashtables), (rnrs)

hashtable must be a mutable hashtable. If size is provided, it must be a nonnegative exact integer.

hashtable-clear! removes all entries from hashtable. If size is provided, the hashtable is reset to the given size, as if newly created by one of the hashtable creation operations with size argument size.

(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 "first")
(hashtable-set! ht p2 "second")
(hashtable-size ht) ⇒ 2
(hashtable-clear! ht)
(hashtable-size ht) ⇒ 0
(hashtable-ref ht p1 #f) ⇒ #f
procedure: (hashtable-keys hashtable)
returns: a vector containing the keys in hashtable
libraries: (rnrs hashtables), (rnrs)

The keys may appear in any order in the returned vector.

(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 "one")
(hashtable-set! ht p2 "two")
(hashtable-set! ht 'q "three")
(hashtable-keys ht) ⇒ #((a . b) q (a . b))

procedure: (hashtable-entries hashtable)
returns: two vectors: one of keys and a second of values
libraries: (rnrs hashtables), (rnrs)

hashtable-entries returns two values. The first is a vector containing the keys in hashtable, and the second is a vector containing the corresponding values. The keys and values may appear in any order, but the order is the same for the keys and for the corresponding values.

(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 "one")
(hashtable-set! ht p2 "two")
(hashtable-set! ht 'q "three")
(hashtable-entries ht) ⇒ #((a . b) q (a . b))
            #("two" "three" "one")

Section 6.14. Enumerations

Enumerations are ordered sets of symbols, typically used to name and manipulate options, as with the buffer modes and file options that may be specified when files are created.

syntax: (define-enumeration name (symbol ...) constructor)
libraries: (rnrs enums), (rnrs)

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

The define-enumeration syntax creates a new enumeration set with the specified symbols in the specified order forming the enumeration’s universe. It defines a new syntactic form named by name that may be used to verify that a symbol is in the universe. If x is in the universe, (name x) evaluates to x. It is a syntax violation if x is not in the universe.

define-enumeration also defines a new syntactic form named by constructor that may be used to create subsets of the enumeration type. If x ... are each in the universe, (constructor x ...) evaluates to an enumeration set containing x ... Otherwise, it is a syntax violation. The same symbol may appear more than once in x ..., but the resulting set contains only one occurrence of the symbol.

(define-enumeration weather-element
  (hot warm cold sunny rainy snowy windy)
weather)
(weather-element hot) ⇒ hot
(weather-element fun) ⇒ syntax violation
(weather hot sunny windy) ⇒ #<enum-set>
(enum-set->list (weather rainy cold rainy)) ⇒ (cold rainy)

procedure: (make-enumeration symbol-list)
returns: an enumeration set
libraries: (rnrs enums), (rnrs)

This procedure creates a new enumeration type whose universe comprises the elements of symbol-list, which must be a list of symbols, in the order of their first appearance in the list. It returns the universe of the new enumeration type as an enumeration set.

(define positions (make-enumeration '(top bottom above top beside))
(enum-set->list positions) ⇒ (top bottom above beside)

procedure: (enum-set-constructor enum-set)
returns: an enumeration-set construction procedure
libraries: (rnrs enums), (rnrs)

This procedure returns a procedure p that may be used to create subsets of the universe of enum-set. p must be passed a list of symbols, and each element of the list must be an element of the universe of enum-set. The enumeration set returned by p contains all and only the symbols in the list it is passed. The value returned by p may contain elements not in enum-set if the universe of enum-set contains those elements.

(define e1 (make-enumeration '(one two three four)))
(define p1 (enum-set-constructor e1))
(define e2 (p1 '(one three)))
(enum-set->list e2) ⇒ (one three)
(define p2 (enum-set-constructor e2))
(define e3 (p2 '(one two four)))
(enum-set->list e3) ⇒ (one two four)

procedure: (enum-set-universe enum-set)
returns: the universe of enum-set, as an enumeration set
libraries: (rnrs enums), (rnrs)

(define e1 (make-enumeration '(a b c a b c d)))
(enum-set->list (enum-set-universe e1)) ⇒ (a b c d)
(define e2 ((enum-set-constructor e1) '(c)))
(enum-set->list (enum-set-universe e2)) ⇒ (a b c d)

procedure: (enum-set->list enum-set)
returns: a list of the elements of enum-set
libraries: (rnrs enums), (rnrs)

The symbols in the resulting list appear in the order given to them when the enumeration type of enum-set was created.

(define e1 (make-enumeration '(a b c a b c d)))
(enum-set->list e1) ⇒ (a b c d)
(define e2 ((enum-set-constructor e1) '(d c a b)))
(enum-set->list e2) ⇒ (a b c d)
procedure: (enum-set-subset? enum-set₁ enum-set₂)
returns: #t if enum-set₁ is a subset of enum-set₂, #f otherwise
libraries: (rnrs enums), (rnrs)

An enumeration set enum-set₁ is a subset of an enumeration set enum-set₂ if and only if
the universe of enum-set₁ is a subset of the universe of enum-set₂ and each element
of enum-set₁ is an element of enum-set₂.

(define e₁ (make-enumeration '(a b c)))
(define e₂ (make-enumeration '(a b c d e)))
(enum-set-subset? e₁ e₂) ⇒ #t
(enum-set-subset? e₂ e₁) ⇒ #f
(define e₃ ((enum-set-constructor e₂) '(a c)))
(enum-set-subset? e₃ e₁) ⇒ #f
(enum-set-subset? e₃ e₂) ⇒ #t

procedure: (enum-set=? enum-set₁ enum-set₂)
returns: #t if enum-set₁ and enum-set₂ are equivalent, #f otherwise
libraries: (rnrs enums), (rnrs)

Two enumeration sets enum-set₁ and enum-set₂ are equivalent if each is a
subset of the other.

(define e₁ (make-enumeration '(a b c d)))
(define e₂ (make-enumeration '(b d c a)))
(enum-set=? e₁ e₂) ⇒ #t
(define e₃ ((enum-set-constructor e₁) '(a c)))
(define e₄ ((enum-set-constructor e₂) '(a c)))
(enum-set=? e₃ e₄) ⇒ #t
(enum-set=? e₃ e₂) ⇒ #f

enum-set=? could be defined in terms of enum-set-subset? as follows.

(define enum-set=?)
  (lambda (e₁ e₂)
    (and (enum-set-subset? e₁ e₂) (enum-set-subset? e₂ e₁)))

procedure: (enum-set-member? symbol enum-set)
returns: #t if symbol is an element of enum-set, #f otherwise
libraries: (rnrs enums), (rnrs)

(define e₁ (make-enumeration '(a b c d e)))
(define e₂ ((enum-set-constructor e₁) '(d b)))
(enum-set-member? 'c e₁) ⇒ #t
(enum-set-member? 'c e₂) ⇒ #f

procedure: (enum-set-union enum-set₁ enum-set₂)
returns: the union of enum-set₁ and enum-set₂
procedure: (enum-set-intersection enum-set₁ enum-set₂)
returns: the intersection of enum-set₁ and enum-set₂
procedure: (enum-set-difference enum-set₁ enum-set₂)
returns: the difference of enum-set₁ and enum-set₂
libraries: (rnrs enums), (rnrs)

enum-set₁ and enum-set₂ must have the same enumeration type. Each procedure returns a new
An enumeration set representing the union, intersection, or difference of the two sets.

```scheme
(define e1 (make-enumeration '(a b c d)))
(define e2 ((enum-set-constructor e1) '(a c)))
(define e3 ((enum-set-constructor e1) '(b c)))
(enum-set->list (enum-set-union e2 e3)) ⇒ (a b c)
(enum-set->list (enum-set-intersection e2 e3)) ⇒ (c)
(enum-set->list (enum-set-difference e2 e3)) ⇒ (a)
(enum-set->list (enum-set-difference e3 e2)) ⇒ (b)
(define e4 (make-enumeration '(b d c a)))
(enum-set-union e1 e4) ⇒ exception: different enumeration types
```

**procedure**: (enum-set-complement enum-set)
**returns**: the complement of enum-set relative to its universe
**libraries**: (rnrs enums), (rnrs)

```scheme
(define el (make-enumeration '(a b c d)))
(enum-set->list (enum-set-complement el)) ⇒ ()
(define e2 ((enum-set-constructor el) '(a c)))
(enum-set->list (enum-set-complement e2)) ⇒ (b d)
```

**procedure**: (enum-set-projection enum-set₁ enum-set₂)
**returns**: the projection of enum-set₁ into the universe of enum-set₂
**libraries**: (rnrs enums), (rnrs)

Any elements of enum-set₁ not in the universe of enum-set₂ are dropped. The result is of the same enumeration type as enum-set₂.

```scheme
(define e1 (make-enumeration '(a b c d)))
(define e2 (make-enumeration '(a b c d e f g)))
(define e3 ((enum-set-constructor e1) '(a d)))
(define e4 ((enum-set-constructor e2) '(a c e g)))
(enum-set->list (enum-set-projection e4 e3)) ⇒ (a c)
(enum-set->list
  (enum-set-union e3
    (enum-set-projection e4 e3))) ⇒ (a c d)
```

**procedure**: (enum-set-indexer enum-set)
**returns**: a procedure that returns the index of a symbol in the universe of enum-set
**libraries**: (rnrs enums), (rnrs)

enum-set-indexer returns a procedure p that, when applied to a symbol in the universe of enum-set, returns the index of the symbol (zero-based) in the ordered set of symbols that form the universe. If applied to a symbol not in the universe, p returns #f.

```scheme
(define e1 (make-enumeration '(a b c d)))
(define e2 ((enum-set-constructor e1) '(a d)))
(define p (enum-set-indexer e2))
(list (p 'a) (p 'c) (p 'e)) ⇒ (0 2 #f)
```
All input and output operations are performed through ports. A port is a pointer into a (possibly infinite) stream of data (often a file), an opening through which programs may draw bytes or characters from the stream or place bytes or characters into the stream. A port may be an input port, an output port, or both simultaneously.

Ports are first-class objects, like any other object in Scheme. Like procedures, ports do not have a printed representation the way strings and numbers do. There are initially three ports: the current input port, current output port, and current error port, which are textual ports connected to the process's standard input, standard output, and standard error streams. Several ways to open new ports are provided.

An input port often points to a finite stream, e.g., an input file stored on disk. If one of the input operations, e.g., get-u8, get-char, or get-datum, is asked to read from a port that has reached the end of a finite stream, it returns a special eof (end of file) object. The predicate eof-object? may be used to determine if the value returned from the input operation is the eof object.

Ports are either binary or textual. A binary port allows a program to read or write 8-bit unsigned bytes, or "octets," from or to the underlying stream. A textual port allows a program to read or write characters.

In many cases, the underlying stream is organized as a sequence of bytes, but these bytes should be treated as encodings for characters. In this case, a textual port may be created with a transcoder to decode bytes to characters (for input) or encode characters to bytes (for output). A transcoder encapsulates a codec that determines how characters are represented as bytes. Three standard codecs are provided: a latin-1 codec, a Unicode utf-8 codec, and a Unicode utf-16 codec. For the latin-1 encoding, each character is represented by exactly one byte. For utf-8, each character is represented by from one to four bytes, and for utf-16, each character is represented by two or four bytes.

A transcoder also encapsulates an eol style that determines whether and how line endings are recognized. If the eol style is none, no line endings are recognized. The six other standard eol styles are the following:

- \( \text{lf} \): line-feed character
- \( \text{cr} \): carriage-return character
- \( \text{nel} \): Unicode next-line character
- \( \text{ls} \): Unicode line-separator character
- \( \text{crlf} \): carriage return followed by line feed, and
- \( \text{crnl} \): carriage return followed by next line

The eol style affects input and output operations differently. For input, any eol style except none causes each of the line-ending characters or two-character sequences to be converted into a single line-feed character. For output, any eol style except none causes line-feed characters to be converted into the specific one- or two-character sequence associated with the eol style. In the input direction, all eol styles except none are equivalent, while in the output direction, the eol styles none and \( \text{lf} \) are equivalent.

In addition to the codec and eol style, a transcoder encapsulates just one other piece of information: an error-handling mode that determines what happens if a decoding or encoding error occurs, i.e., if a sequence of bytes cannot be converted to a character with the encapsulated codec in the input direction or a character cannot be converted to a sequence of bytes with the encapsulated codec in the output direction. The error-handling mode is ignore, raise, or replace. If the error-handling mode is ignore, the offending sequence of bytes or the character is ignored. If the error-handling mode is raise, an exception with condition type i/o-decoding or i/o-encoding is raised; in the input direction, the port is positioned beyond the sequence of bytes. If the error-handling mode is replace, a replacement character or character encoding is produced: in the input direction, the replacement character is U+FFFD, while in the output
direction, the replacement is either the encoding of U+FFFD for utf-8 and utf-16 codecs or the encoding of the question-mark character (?) for the latin-1 codec.

A port may be buffered for efficiency, to eliminate the overhead of a call into the operating system for each byte or character. Three standard buffer modes are supported: block, line, and none. With block buffering, input is drawn from a stream and output is sent to the stream in chunks of some implementation-dependent size. With line buffering, buffering is performed on a line-by-line basis or on some other implementation-dependent basis. Line buffering is typically distinguished from block buffering only for textual output ports; there are no line divisions in binary ports, and input is likely to be drawn from a stream as it becomes available. With buffer-mode none, no buffering is performed, so output is sent immediately to the stream and input is drawn only as needed.

The remainder of this chapter covers operations on transcoders, file ports, standard ports, string and byteword ports, custom ports, general port operations, input operations, output operations, convenience I/O, filesystem operations, and conversions between bytevectors and strings.

Section 7.1. Transcoders

As described above, transcoders encapsulate three values: a codec, an eol style, and an error-handling mode. This section describes the procedures that create or operate on transcoders and the values that transcoders encapsulate.

**procedure**: `(make-transcoder codec)`

**returns**: a transcoder encapsulating codec

**libraries**: (rnrs io ports), (rnrs)

**eol-style** must be a valid eol-style symbol (lf, cr, nel, ls, crlf, crnel, or none); it defaults to the native eol-style for the platform. **error-handling-mode** must be a valid error-handling-mode symbol (ignore, raise, or replace) and defaults to replace.

**procedure**: `(transcoder-codec transcoder)`

**returns**: the codec encapsulated in transcoder

**procedure**: `(transcoder-eol-style transcoder)`

**returns**: the eol-style symbol encapsulated in transcoder

**procedure**: `(transcoder-error-handling-mode transcoder)`

**returns**: the error-handling-mode symbol encapsulated in transcoder

**libraries**: (rnrs io ports), (rnrs)

**procedure**: `(native-transcoder)`

**returns**: the native transcoder

**libraries**: (rnrs io ports), (rnrs)

The native transcoder is implementation-dependent and may vary by platform or locale.

**procedure**: `(latin-1-codec)`

**returns**: a codec for ISO 8859-1 (Latin 1) character encodings

**procedure**: `(utf-8-codec)`

**returns**: a codec for Unicode UTF-8 character encodings

**procedure**: `(utf-16-codec)`

**returns**: a codec for Unicode UTF-16 character encodings

**libraries**: (rnrs io ports), (rnrs)
**Section 7.2. Opening Files**

The procedures in this section are used for opening file ports. Procedures for opening other kinds of ports, e.g., string ports or custom ports, are described in subsequent sections.

Each of the file-open operations accepts a path argument that names the file to be opened. It must be a string or some other implementation-dependent value that names a file.

Some of the file-open procedures accept optional options, b-mode, and ?transcoder arguments. options must be an enumeration set over the symbols constituting valid file options described in the file-options entry below, and it defaults to the value of (file-options). b-mode must be a valid buffer mode described in the buffer-mode entry below, and it defaults to block. ?transcoder must be a transcoder or #f; if it is a transcoder, the open operation returns a transcoded port for the underlying binary file, while if it is #f (the default), the open operation returns a binary port.

Binary ports created by the procedures in this section support the port-position and set-port-position! operations. Whether textual ports created by the procedures in this section support these operations is implementation-dependent.
File-options enumeration sets may be passed to file-open operations to control aspects of the open operation. There are three standard file options: no-create, no-fail, and no-truncate, which affect only file-open operations that create output (including input/output) ports.

With the default file options, i.e., the value of (file-options), when a program attempts to open a file for output, an exception is raised with condition type i/o-file-already-exists if the file already exists, and the file is created if it does not already exist. If the no-fail option is included, no exception is raised if the file already exists; instead, the file is opened and truncated to zero length. If the no-create option is included, the file is not created if it does not exist; instead, an exception is raised with condition type i/o-file-does-not-exist. The no-create option implies the no-fail option. The no-truncate option is relevant only if the no-fail option is included or implied, in which case if an existing file is opened, it is not truncated, but the port's position is still set to the beginning of the file.

It is perhaps easier to imagine that the default file options are the imaginary option symbols create, fail-if-exists, and truncate; no-create removes create, no-fail removes fail-if-exists, and no-truncate removes truncate.

Implementations may support additional file option symbols. Chez Scheme, for example, supports options that control whether the file is or should be compressed, whether it is locked for exclusive access, and what permissions are given to the file if it is created [9].

**syntax:** (buffer-mode symbol)
**returns:** symbol
**libraries:** (rnrs io ports), (rnrs)

symbol must be one of the symbols block, line, or none. The expression (buffer-mode symbol) is equivalent to the expression (quote symbol) except that the former checks at expansion time that symbol is one of the buffer-mode symbols. The buffer-mode syntax provides useful documentation as well.

(buffer-mode block) ⇒ block
(buffer-mode cushion) ⇒ syntax violation

**syntax:** (buffer-mode? obj)
**returns:** #t if obj is a valid buffer mode, #f otherwise
**libraries:** (rnrs io ports), (rnrs)

(buffer-mode? 'block) ⇒ #t
(buffer-mode? 'line) ⇒ #t
(buffer-mode? 'none) ⇒ #t
(buffer-mode? 'something-else) ⇒ #f

**procedure:** (open-file-input-port path)
**procedure:** (open-file-input-port path options)
**procedure:** (open-file-input-port path options b-mode)
**procedure:** (open-file-input-port path options b-mode ?transcoder)
**returns:** a new input port for the named file
**libraries:** (rnrs io ports), (rnrs)

If ?transcoder is present and not #f, it must be a transcoder, and this procedure returns a textual input port whose transcoder is ?transcoder. Otherwise, this procedure returns a binary input port. See the lead-in to this section for a description of the constraints on and effects of the other arguments.

**procedure:** (open-file-output-port path)
The Scheme Programming Language, 4th Edition

procedure: (open-file-output-port path options)
procedure: (open-file-output-port path options b-mode)
procedure: (open-file-output-port path options b-mode ?transcoder)
returns: a new output port for the named file
libraries: (rnrs io ports), (rnrs)

If ?transcoder is present and not #f, it must be a transcoder, and this procedure returns a textual output port whose transcoder is ?transcoder. Otherwise, this procedure returns a binary output port. See the lead-in to this section for a description of the constraints on and effects of the other arguments.

procedure: (open-file-input/output-port path)
procedure: (open-file-input/output-port path options)
procedure: (open-file-input/output-port path options b-mode)
procedure: (open-file-input/output-port path options b-mode ?transcoder)
returns: a new input/output port for the named file
libraries: (rnrs io ports), (rnrs)

If ?transcoder is present and not #f, it must be a transcoder, and this procedure returns a textual input/output port whose transcoder is ?transcoder. Otherwise, this procedure returns a binary input/output port. See the lead-in to this section for a description of the constraints on and effects of the other arguments.

Section 7.3. Standard Ports

The procedures described in this section return ports that are attached to a process's standard input, standard output, and standard error streams. The first set returns "ready-made" textual ports with implementation-dependent transcoders (if any) and buffer modes. The second set creates fresh binary ports and can be used either for binary input/output or, with the help of transcoded-port, for textual input/output with program-supplied transcoders and buffer modes.

procedure: (current-input-port)
returns: the current input port
procedure: (current-output-port)
returns: the current output port
procedure: (current-error-port)
returns: the current error port
libraries: (rnrs io ports), (rnrs io simple), (rnrs)

The current-input, current-output, and current-error ports return pre-built textual ports that are initially associated with a process's standard input, standard output, and standard error streams.

The values returned by current-input-port and current-output-port can be altered temporarily by the convenience I/O procedures with-input-from-file and with-output-to-file (Section 7.9).

procedure: (standard-input-port)
returns: a fresh binary input port connected to the standard input stream
procedure: (standard-output-port)
returns: a fresh binary output port connected to the standard output stream
procedure: (standard-error-port)
returns: a fresh binary output port connected to the standard error stream
libraries: (rnrs io ports), (rnrs)

Because ports may be buffered, confusion can result if operations on more than one port attached to one of a process's standard streams are interleaved. Thus, these procedures are typically appropriate only when a
program no longer needs to use any existing ports attached to the standard streams.

Section 7.4. String and Bytevector Ports

The procedures in this section allow bytevectors and strings to be used as input or output streams.

Binary ports created by the procedures in this section support the `port-position` and `set-port-position!` operations. Whether textual ports created by the procedures in this section support these operations is implementation-dependent.

```
procedure: (open-bytevector-input-port bytevector)
procedure: (open-bytevector-input-port bytevector ?transcoder)
returns: a new input port that draws input from bytevector
libraries: (rnrs io ports), (rnrs)
```

If `?transcoder` is present and not `#f`, it must be a transcoder, and this procedure returns a textual input port whose transcoder is `?transcoder`. Otherwise, this procedure returns a binary input port.

The effect of modifying `bytevector` after this procedure is called is unspecified.

```
(let ([ip (open-bytevector-input-port #vu8(1 2))])
  (let* ([x1 (get-u8 ip)] [x2 (get-u8 ip)] [x3 (get-u8 ip)])
    (list x1 x2 (eof-object? x3)))) ⇒ (1 2 #t)
```

There is no need to close a bytevector port; it's storage will be reclaimed automatically when it is no longer needed, as with any other object, and an open bytevector port does not tie up any operating system resources.

```
procedure: (open-string-input-port string)
returns: a new textual input port that draws input from string
libraries: (rnrs io ports), (rnrs)
```

The effect of modifying `string` after this procedure is called is unspecified. The new port may or may not have a transcoder, and if it does, the transcoder is implementation-dependent. While not required, implementations are encouraged to support `port-position` and `set-port-position!` for string ports.

```
(get-line (open-string-input-port "hi.\nwhat's up?\n")) ⇒ "hi."
```

There is no need to close a string port; it's storage will be reclaimed automatically when it is no longer needed, as with any other object, and an open string port does not tie up any operating system resources.

```
procedure: (open-bytevector-output-port)
procedure: (open-bytevector-output-port ?transcoder)
returns: two values, a new output port and an extraction procedure
libraries: (rnrs io ports), (rnrs)
```

If `?transcoder` is present and not `#f`, it must be a transcoder, and the port value is a textual output port whose transcoder is `?transcoder`. Otherwise, the port value is a binary output port.

The extraction procedure is a procedure that, when called without arguments, creates a bytevector containing the accumulated bytes in the port, clears the port of its accumulated bytes, resets its position to zero, and returns the bytevector. The accumulated bytes include any bytes written beyond the end of the current position, if the position has been set back from its maximum extent.
(let-values ([(op g) (open-bytevector-output-port)]
  (put-u8 op 15)
  (put-u8 op 73)
  (put-u8 op 115)
  (set-port-position! op 2)
  (let ([bv1 (g)])
    (put-u8 op 27)
    (list bv1 (g)))))) => (#vu8(15 73 115) #vu8(27))

There is no need to close a bytevector port; its storage will be reclaimed automatically when it is no longer needed, as with any other object, and an open bytevector port does not tie up any operating system resources.

**procedure:** (open-string-output-port)
**returns:** two values, a new textual output port and an extraction procedure
**libraries:** (rnrs io ports), (rnrs)

The extraction procedure is a procedure that, when called without arguments, creates a string containing the accumulated characters in the port, clears the port of its accumulated characters, resets its position to zero, and returns the string. The accumulated characters include any characters written beyond the end of the current position, if the position has been set back from its maximum extent. While not required, implementations are encouraged to support `port-position` and `set-port-position!` for string ports.

(let-values ([(op g) (open-string-output-port)]
  (put-string op "some data")
  (let ([str1 (g)])
    (put-string op "new stuff")
    (list str1 (g)))))) => ("some data" "new stuff")

There is no need to close a string port; its storage will be reclaimed automatically when it is no longer needed, as with any other object, and an open string port does not tie up any operating system resources.

**procedure:** (call-with-bytevector-output-port procedure)
**procedure:** (call-with-bytevector-output-port procedure ?transcoder)
**returns:** a bytevector containing the accumulated bytes
**libraries:** (rnrs io ports), (rnrs)

If `?transcoder` is present and not `#f`, it must be a transcoder, and `procedure` is called with a textual bytevector output port whose transcoder is `?transcoder`. Otherwise, `procedure` is called with a binary bytevector output port. If `procedure` returns, a bytevector containing the bytes accumulated in the port is created, the accumulated bytes are cleared from the port, the port’s position is reset to zero, and the bytevector is returned from `call-with-bytevector-output-port`. These actions occur each time `procedure` returns, if it returns multiple times due to the invocation of a continuation created while `procedure` is active.

(let ((tx (make-transcoder (latin-1-codec) (eol-style lf)
  (error-handling-mode replace)))))
  (call-with-bytevector-output-port
    (lambda (p) (put-string p "abc")
      tx))))) => #vu8(97 98 99)

**procedure:** (call-with-string-output-port procedure)
**returns:** a string containing the accumulated characters
**libraries:** (rnrs io ports), (rnrs)
procedure is called with one argument, a string output port. If procedure returns, a string containing the characters accumulated in the port is created, the accumulated characters are cleared from the port, the port’s position is reset to zero, and the string is returned from call-with-string-output-port. These actions occur each time procedure returns, if it returns multiple times due to the invocation of a continuation created while procedure is active.

call-with-string-output-port can be used along with put-datum to define a procedure, object->string, that returns a string containing the printed representation of an object.

(define (object->string x)
  (call-with-string-output-port
   (lambda (p) (put-datum p x))))

(object->string (cons 'a '(b c))) => "(a b c)"

Section 7.5. Opening Custom Ports

procedure: (make-custom-binary-input-port id r! gp sp! close)
returns: a new custom binary input port
procedure: (make-custom-binary-output-port id w! gp sp! close)
returns: a new custom binary output port
procedure: (make-custom-binary-input/output-port id r! w! gp sp! close)
returns: a new custom binary input/output port
libraries: (rnrs io ports), (rnrs)

These procedures allow programs to create ports from arbitrary byte streams. id must be a string naming the new port; the name is used for informational purposes only, and an implementation may choose to include it in the printed syntax, if any, of a custom port. r! and w! must be procedures, while gp, sp!, and close must each be a procedure or #f. These arguments are described below.

r!
is called to draw input from the custom port, e.g., to support get-u8 or get-bytevector-n. It is called with three arguments: bytevector, start, and n. start will be a nonnegative exact integer, n will be a positive exact integer, and the sum of start and n will not exceed the length of bytevector. If the byte stream is at end of file, r! should return exact 0. Otherwise, it should read at least one and at most n bytes from the stream, store these bytes in consecutive locations of bytevector starting at start, and return as an exact positive integer the number of bytes actually read.

w!
is called to send output to the port, e.g., to support put-u8 or put-bytevector. It is called with three arguments: bytevector, start, and n. start and n will be nonnegative exact integers, and the sum of start and n will not exceed the length of bytevector. w! should write up to n consecutive bytes from bytevector starting at start and return, as an exact nonnegative integer, the number of bytes actually written.

gp
is called to query the port’s position. If it is #f, the port will not support port-position. If it is not #f, it will be passed zero arguments and should return the current position as a displacement in bytes from the start of the byte stream as an exact nonnegative integer.

sp!
is called to set the port’s position. If it is #f, the port will not support set-port-position!. If it is not #f, it will be passed one argument, an exact nonnegative integer representing the new position as a displacement in bytes from the start of the byte stream, and it should set the position to this value.
is called to close the byte stream. If it is #f, no action will be taken to close the byte stream when the new port is closed. If it is not #f, it will be passed zero arguments and should take whatever actions are necessary to close the byte stream.

If the new port is an input/output port and does not provide either a gp or sp! procedure, it may not be possible for the implementation to position the port properly if an output operation occurs after an input operation, due to input buffering that must be done to support lookahead-u8 and is often done anyway for efficiency. For the same reason, a call to port-position after an input operation may not return an accurate position if the sp! procedure is not provided. Thus, programs that create custom binary input/output ports should generally provide both gp and sp! procedures.

These procedures allow programs to create ports from arbitrary character streams. id must be a string naming the new port; the name is used for informational purposes only, and an implementation may choose to include it in the printed syntax, if any, of a custom port. r! and w! must be procedures, while gp, sp!, and close must each be a procedure or #f. These arguments are described below.

r!  
is called to draw input from the port, e.g., to support get-char or get-string-n. It is called with three arguments: string, start, and n. start will be a nonnegative exact integer, n will be a positive exact integer, and the sum of start and n will not exceed the length of string. If the character stream is at end of file, r! should return exact 0. Otherwise, it should read at least one and at most n characters from the stream, store these characters in consecutive locations of string starting at start, and return as an exact positive integer the number of characters actually read.

w!  
is called to send output to the port, e.g., to support put-char or put-string. It is called with three arguments: string, start, and n. start and n will be nonnegative exact integers, and the sum of start and n will not exceed the length of string. w! should write up to n consecutive characters from string starting at start and return, as an exact nonnegative integer, the number of characters actually written.

gp  
is called to query the port's position. If it is #f, the port will not support port-position. If it is not #f, it will be passed zero arguments and should return the current position, which may be an arbitrary value.

sp!  
is called to set the port's position. If it is #f, the port will not support set-port-position!. If it is not #f, it will be passed one argument, pos, a value representing the new position. If pos is the result of a previous call to gp, sp! should set the position to pos.

close  
is called to close the character stream. If it is #f, no action will be taken to close the character stream when the new port is closed. If it is not #f, it will be passed zero arguments and should take whatever actions are necessary to close the character stream.

If the new port is an input/output port, it may not be possible for the implementation to position the port properly if an output operation occurs after an input operation, even if the gp and sp! procedures are provided, due to input buffering that must be done to support lookahead-char and is often done anyway for efficiency. Since the representations of port positions are not specified, it is not possible for the
implementation to adjust the \textit{gp} return value to account for the number of buffered characters. For the same reason, a call to \texttt{port-position} after an input operation may not return an accurate position, even if the \texttt{sp!} procedure is provided.

It should, however, be possible to perform output reliably after reading if the position is reset to the starting position. Thus, programs that create custom textual input/output ports should generally provide both \texttt{gp} and \texttt{sp!} procedures, and consumers of these ports should obtain the starting position via \texttt{port-position} before any input operations and reset the position back to the starting position before doing any output operations.

\section*{Section 7.6. Port Operations}

This section describes a variety of operations on ports that do not directly involve either reading from or writing to a port. The input and output operations are described in subsequent sections.

\begin{verbatim}
procedure: (port? obj)
returns: #t if \textit{obj} is a port, #f otherwise
libraries: (rnrs io ports), (rnrs)

procedure: (input-port? obj)
returns: #t if \textit{obj} is an input or input/output port, #f otherwise
procedure: (output-port? obj)
returns: #t if \textit{obj} is an output or input/output port, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs)

procedure: (binary-port? obj)
returns: #t if \textit{obj} is a binary port, #f otherwise
procedure: (textual-port? obj)
returns: #t if \textit{obj} is a textual port, #f otherwise
libraries: (rnrs io ports), (rnrs)

procedure: (close-port port)
returns: unspecified
libraries: (rnrs io ports), (rnrs)

procedure: (transcoded-port binary-port transcoder)
returns: a new textual port with the same byte stream as \textit{binary-port}
libraries: (rnrs io ports), (rnrs)
\end{verbatim}

If \textit{port} is not already closed, \texttt{close-port} closes it, first flushing any buffered bytes or characters to the underlying stream if the port is an output port. Once a port has been closed, no more input or output operations may be performed on the port. Because the operating system may place limits on the number of file ports open at one time or restrict access to an open file, it is good practice to close any file port that will no longer be used for input or output. If the port is an output port, closing the port explicitly also ensures that buffered data is written to the underlying stream. Some Scheme implementations close file ports automatically after they become inaccessible to the program or when the Scheme program exits, but it is best to close file ports explicitly whenever possible. Closing a port that has already been closed has no effect.

This procedure returns a new textual port with transcoder \textit{transcoder} and the same underlying byte stream as \textit{binary-port}, positioned at the current position of \textit{binary-port}.

As a side effect of creating the textual port, \textit{binary-port} is closed to prevent read or write operations on \textit{binary-port} from interfering with read and write operations on the new textual port. The underlying byte stream remains open, however, until the textual port is closed.
procedure: (port-transcoder port)
returns: the transcoder associated with port if any, #f otherwise
libraries: (rnrs io ports), (rnrs)

This procedure always returns #f for binary ports and may return #f for some textual ports.

procedure: (port-position port)
returns: the port's current position

procedure: (port-has-port-position? port)
returns: #t if the port supports port-position, #f otherwise
libraries: (rnrs io ports), (rnrs)

A port may allow queries to determine its current position in the underlying stream of bytes or characters. If so, the procedure port-has-port-position? returns #t and port-position returns the current position. For binary ports, the position is always an exact nonnegative integer byte displacement from the start of the byte stream. For textual ports, the representation of a position is unspecified; it may not be an exact nonnegative integer and, even if it is, it may not represent either a byte or character displacement in the underlying stream. The position may be used at some later time to reset the position if the port supports set-port-position!. If port-position is called on a port that does not support it, an exception with condition type &assertion is raised.

procedure: (set-port-position! port pos)
returns: unspecified

procedure: (port-has-set-port-position!? port)
returns: #t if the port supports set-port-position!, #f otherwise
libraries: (rnrs io ports), (rnrs)

A port may allow its current position to be moved directly to a different position in the underlying stream of bytes or characters. If so, the procedure port-has-set-port-position!? returns #t and set-port-position! changes the current position. For binary ports, the position pos must be an exact nonnegative integer byte displacement from the start of the byte stream. For textual ports, the representation of a position is unspecified, as described in the entry for port-position above, but pos must be an appropriate position for the textual port, which is usually guaranteed to be the case only if it was obtained from a call to port-position on the same port. If set-port-position! is called on a port that does not support it, an exception with condition type &assertion is raised.

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

procedure: (call-with-port port procedure)
returns: the values returned by procedure
libraries: (rnrs io ports), (rnrs)

call-with-port calls procedure with port as the only argument. If procedure returns, call-with-port closes the port and returns the values returned by procedure. If procedure does not return, an implementation is free to close the
port only if it can prove that the output port is no longer accessible.

The example below copies the contents of infile to outfile, overwriting outfile if it exists. Unless an error occurs, the ports are closed after the copy has been completed.

```scheme
(call-with-port (open-file-input-port "infile" (file-options)
  (buffer-mode block) (native-transcoder))
  (lambda (ip)
    (call-with-port (open-file-output-port "outfile"
      (file-options no-fail)
      (buffer-mode block)
      (native-transcoder))
      (lambda (op)
        (do ([c (get-char ip) (get-char ip)])
          ((eof-object? c))
          (put-char op c))))))
```

A definition of `call-with-port` is given on page 135.

**procedure**: `(output-port-buffer-mode port)`
**returns**: the symbol representing the buffer mode of `port`
**libraries**: `(rnrs io ports), (rnrs)

### Section 7.7. Input Operations

Procedures whose primary purpose is to read data from an input port are described in this section, along with related procedures for recognizing or creating end-of-file (eof) objects.

**procedure**: `(eof-object? obj)`
**returns**: `#t` if `obj` is an eof object, `#f` otherwise
**libraries**: `(rnrs io ports), (rnrs io simple), (rnrs)

The end-of-file object is returned by input operations, e.g., `get-datum`, when an input port has reached the end of input.

**procedure**: `(eof-object)`
**returns**: the eof object
**libraries**: `(rnrs io ports), (rnrs io simple), (rnrs)

`(eof-object? (eof-object)) ⇒ #t`

**procedure**: `(get-u8 binary-input-port)`
**returns**: the next byte from `binary-input-port`, or the eof object
**libraries**: `(rnrs io ports), (rnrs)

If `binary-input-port` is at end of file, the eof object is returned. Otherwise, the next available byte is returned as an unsigned 8-bit quantity, i.e., an exact unsigned integer less than or equal to 255, and the port's position is advanced one byte.

**procedure**: `(lookahead-u8 binary-input-port)`
**returns**: the next byte from `binary-input-port`, or the eof object
**libraries**: `(rnrs io ports), (rnrs)"
If \texttt{binary-input-port} is at end of file, the \texttt{eof} object is returned. Otherwise, the next available byte is returned as an unsigned 8-bit quantity, i.e., an exact unsigned integer less than or equal to 255. In contrast to \texttt{get-u8}, \texttt{lookahead-u8} does not consume the byte it reads from the port, so if the next operation on the port is a call to \texttt{lookahead-u8} or \texttt{get-u8}, the same byte is returned.

\textbf{procedure}: (get-bytevector-n \texttt{binary-input-port} \texttt{n})
\textbf{returns}: a nonempty bytevector containing up to \texttt{n} bytes, or the \texttt{eof} object
\textbf{libraries}: (rnrs io ports), (rnrs)

\texttt{n} must be an exact nonnegative integer. If \texttt{binary-input-port} is at end of file, the \texttt{eof} object is returned. Otherwise, \texttt{get-bytevector-n} reads (as if with \texttt{get-u8}) as many bytes, up to \texttt{n}, as are available before the port is at end of file, and returns a new (nonempty) bytevector containing these bytes. The port’s position is advanced past the bytes read.

\textbf{procedure}: (get-bytevector-n! \texttt{binary-input-port} \texttt{bytevector} \texttt{start} \texttt{n})
\textbf{returns}: the count of bytes read or the \texttt{eof} object
\textbf{libraries}: (rnrs io ports), (rnrs)

\texttt{start} and \texttt{n} must be exact nonnegative integers, and the sum of \texttt{start} and \texttt{n} must not exceed the length of \texttt{bytevector}.

If \texttt{binary-input-port} is at end of file, the \texttt{eof} object is returned. Otherwise, \texttt{get-bytevector-n!} reads (as if with \texttt{get-u8}) as many bytes, up to \texttt{n}, as are available before the port is at end of file, stores the bytes in consecutive locations of \texttt{bytevector} starting at \texttt{start}, and returns the count of bytes read as an exact positive integer. The port’s position is advanced past the bytes read.

\textbf{procedure}: (get-bytevector-some \texttt{binary-input-port})
\textbf{returns}: a nonempty bytevector or the \texttt{eof} object
\textbf{libraries}: (rnrs io ports), (rnrs)

If \texttt{binary-input-port} is at end of file, the \texttt{eof} object is returned. Otherwise, \texttt{get-bytevector-some} reads (as if with \texttt{get-u8}) at least one byte and possibly more, and returns a bytevector containing these bytes. The port’s position is advanced past the bytes read. The maximum number of bytes read by this operation is implementation-dependent.

\textbf{procedure}: (get-bytevector-all \texttt{binary-input-port})
\textbf{returns}: a nonempty bytevector or the \texttt{eof} object
\textbf{libraries}: (rnrs io ports), (rnrs)

If \texttt{binary-input-port} is at end of file, the \texttt{eof} object is returned. Otherwise, \texttt{get-bytevector-all} reads (as if with \texttt{get-u8}) all of the bytes available before the port is at end of file and returns a bytevector containing these bytes. The port’s position is advanced past the bytes read.

\textbf{procedure}: (get-char \texttt{textual-input-port})
\textbf{returns}: the next character from \texttt{textual-input-port}, or the \texttt{eof} object
\textbf{libraries}: (rnrs io ports), (rnrs)

If \texttt{textual-input-port} is at end of file, the \texttt{eof} object is returned. Otherwise, the next available character is returned and the port’s position is advanced one character. If \texttt{textual-input-port} is a transcoded port, the position in the underlying byte stream may advance by more than one byte.

\textbf{procedure}: (lookahead-char \texttt{textual-input-port})
\textbf{returns}: the next character from \texttt{textual-input-port}, or the \texttt{eof} object
\textbf{libraries}: (rnrs io ports), (rnrs)
If `textual-input-port` is at end of file, the eof object is returned. Otherwise, the next available character is returned. In contrast to `get-char`, `lookahead-char` does not consume the character it reads from the port, so if the next operation on the port is a call to `lookahead-char` or `get-char`, the same character is returned.

`lookahead-char` is provided for applications requiring one character of lookahead. The procedure `get-word` defined below returns the next word from a textual input port as a string, where a word is defined to be a sequence of alphabetic characters. Since `get-word` does not know until it sees one character beyond the word that it has read the entire word, it uses `lookahead-char` to determine the next character and `get-char` to consume the character.

```
(define get-word
  (lambda (p)
    (list->string
      (let f ()
        (let ([c (lookahead-char p)])
          (cond
            [(eof-object? c) '()]
            [(char-alphabetic? c) (get-char p) (cons c (f))]
            [else '()])))))))
```

**procedure**: `(get-string-n textual-input-port n)`  
**returns**: a nonempty string containing up to `n` characters, or the eof object  
**libraries**: `(rnrs io ports), (rnrs)`

`n` must be an exact nonnegative integer. If `textual-input-port` is at end of file, the eof object is returned. Otherwise, `get-string-n` reads (as if with `get-char`) as many characters, up to `n`, as are available before the port is at end of file, and returns a new (nonempty) string containing these characters. The port's position is advanced past the characters read.

```
(define get-string-n!
  (lambda (p string start n)
    (let ([sip (open-string-input-port (string c))])
      (get-string-n! sip s i 1)
      ; return unspecified values:
      (if #f #f))))
```

**procedure**: `(get-string-n! textual-input-port string start n)`  
**returns**: the count of characters read or the eof object  
**libraries**: `(rnrs io ports), (rnrs)`

`start` and `n` must be exact nonnegative integers, and the sum of `start` and `n` must not exceed the length of `string`.

If `textual-input-port` is at end of file, the eof object is returned. Otherwise, `get-string-n!` reads (as if with `get-char`) as many characters, up to `n`, as are available before the port is at end of file, stores the characters in consecutive locations of `string` starting at `start`, and returns the count of characters read as an exact positive integer. The port's position is advanced past the characters read.

`get-string-n!` may be used to implement `string-set!` and `string-fill!`, as illustrated below, although this is not its primary purpose.

```
(define string-set!
  (lambda (s i c)
    (let ([sip (open-string-input-port (string c))])
      (get-string-n! sip s i 1)
      ; return unspecified values:
      (if #f #f)))

(define string-fill!
  (lambda (s c)
(let ([n (string-length s)])
  (let ([sip (open-string-input-port (make-string n c))])
    (get-string-n! sip s 0 n)
    ; return unspecified values:
    (if #f #f)))))

(let ([x (make-string 3)])
  (string-fill! x #\-)
  (string-set! x 2 #\))
  (string-set! x 0 #\;)
  x)

procedure: (get-string-all textual-input-port)
returns: a nonempty string or the eof object
libraries: (rnrs io ports), (rnrs)

If textual-input-port is at end of file, the eof object is returned. Otherwise, get-string-all reads (as if with get-char) all of the characters available before the port is at end of file and returns a string containing these characters. The port’s position is advanced past the characters read.

procedure: (get-line textual-input-port)
returns: a string or the eof object
libraries: (rnrs io ports), (rnrs)

If textual-input-port is at end of file, the eof object is returned. Otherwise, get-line reads (as if with get-char) all of the characters available before the port is at end of file or a line-feed character has been read and returns a string containing all but the line-feed character of the characters read. The port’s position is advanced past the characters read.

(let ([sip (open-string-input-port "one\ntwo

 one (two)\n")]
  (let* ([s1 (get-line sip)] [s2 (get-line sip)])
    (list s1 s2 (port-eof? sip)))) ⇒ ("one" "two" #t)

(let ([sip (open-string-input-port "one\ntwo")])
  (let* ([s1 (get-line sip)] [s2 (get-line sip)])
    (list s1 s2 (port-eof? sip)))) ⇒ ("one" "two" #t)

procedure: (get-datum textual-input-port)
returns: a Scheme datum object or the eof object
libraries: (rnrs io ports), (rnrs)

This procedure scans past whitespace and comments to find the start of the external representation of a datum. If textual-input-port reaches end of file before the start of the external representation of a datum is found, the eof object is returned.

Otherwise, get-datum reads as many characters as necessary, and no more, to parse a single datum, and returns a newly allocated object whose structure is determined by the external representation. The port’s position is advanced past the characters read. If an end-of-file is reached before the external representation of the datum is complete, or an unexpected character is read, an exception is raised with condition types &lexical and i/o-read.

(let ([sip (open-string-input-port "; a\n\n one (two)\n")])
  (let* ([x1 (get-datum sip)]
    [cl (lookahead-char sip)])

procedure: (port-eof? input-port)
returns: #t if input-port is at end-of-file, #f otherwise
libraries: (rnrs io ports), (rnrs)

This procedure is similar to lookahead-u8 on a binary input port or lookahead-char on a textual input port, except that instead of returning the next byte/character or eof object, it returns a boolean value to indicate whether the value would be the eof object.

Section 7.8. Output Operations

Procedures whose primary purpose is to send data to an output port are described in this section.

procedure: (put-u8 binary-output-port octet)
returns: unspecified
libraries: (rnrs io ports), (rnrs)

octet must be an exact nonnegative integer less than or equal to 255. This procedure writes octet to binary-output-port, advancing the port's position by one byte.

procedure: (put-bytevector binary-output-port bytevector)
procedure: (put-bytevector binary-output-port bytevector start)
procedure: (put-bytevector binary-output-port bytevector start n)
returns: unspecified
libraries: (rnrs io ports), (rnrs)

start and n must be nonnegative exact integers, and the sum of start and n must not exceed the length of bytevector. If not supplied, start defaults to zero and n defaults to the difference between the length of bytevector and start.

This procedure writes the n bytes of bytevector starting at start to the port and advances the its position past the end of the bytes written.

procedure: (put-char textual-output-port char)
returns: unspecified
libraries: (rnrs io ports), (rnrs)

This procedure writes char to textual-output-port, advancing the port's position by one character. If textual-output-port is a transcoded port, the position in the underlying byte stream may advance by more than one byte.

procedure: (put-string textual-output-port string)
procedure: (put-string textual-output-port string start)
procedure: (put-string textual-output-port string start n)
returns: unspecified
libraries: (rnrs io ports), (rnrs)

start and n must be nonnegative exact integers, and the sum of start and n must not exceed the length of string. If not supplied, start defaults to zero and n defaults to the difference between the length of string and start.
This procedure writes the $n$ characters of $string$ starting at $start$ to the port and advances the its position past the end of the characters written.

**procedure**: (put-datum textual-output-port obj)
**returns**: unspecified
**libraries**: (rnrs io ports), (rnrs)

This procedure writes an external representation of $obj$ to textual-output-port. If $obj$ does not have an external representation as a datum, the behavior is unspecified. The precise external representation is implementation-dependent, but when $obj$ does have an external representation as a datum, put-datum should produce a sequence of characters that can later be read by get-datum as an object equivalent (in the sense of equal?) to $obj$. See Section 12.5 for an implementation of put-datum, write, and display.

**procedure**: (flush-output-port output-port)
**returns**: unspecified
**libraries**: (rnrs io ports), (rnrs)

This procedure forces any bytes or characters in the buffer associated with $output-port$ to be sent immediately to the underlying stream.

## Section 7.9. Convenience I/O

The procedures in this section are referred to as "convenience" I/O operators because they present a somewhat simplified interface for creating and interacting with textual ports. They also provide backward compatibility with the Revised$^5$ Report, which did not support separate binary and textual I/O.

The convenience input/output procedures may be called with or without an explicit port argument. If called without an explicit port argument, the current input or output port is used, as appropriate. For example, (read-char) and (read-char (current-input-port)) both return the next character from the current input port.

**procedure**: (open-input-file path)
**returns**: a new input port
**libraries**: (rnrs io simple), (rnrs)

$path$ must be a string or some other implementation-dependent value that names a file.

open-input-file creates a new textual input port for the file named by $path$, as if by open-file-input-port with default options, an implementation-dependent buffer mode, and an implementation-dependent transcoder.

The following shows the use of open-input-file, read, and close-port in an expression that gathers a list of objects from the file named by "myfile.ss."

```
(let ([p (open-input-file "myfile.ss")])
  (let f ([x (read p)])
    (if (eof-object? x)
        (begin
          (close-port p)
          '())
        (cons x (f (read p))))))
```

**procedure**: (open-output-file path)
**returns**: a new output port
**libraries**: (rnrs io simple), (rnrs)
**open-output-file** creates a new output port for the file named by *path*, as if by **open-file-output-port** with default options, an implementation-dependent buffer mode, and an implementation-dependent transcoder.

The following shows the use of **open-output-file** to write a list of objects (the value of **list-to-be-printed**), separated by newlines, to the file named by "myfile.ss."

```scheme
(let ([p (open-output-file "myfile.ss")])
  (let f ([ls list-to-be-printed])
    (if (not (null? ls))
      (begin
        (write (car ls) p)
        (newline p)
        (f (cdr ls)))))
  (close-port p))
```

**procedure**: (call-with-input-file *path* procedure)

**returns**: the values returned by **procedure**

**libraries**: (rnrs io simple), (rnrs)

*path* must be a string or some other implementation-dependent value that names a file. **procedure** should accept one argument.

**call-with-input-file** creates a new input port for the file named by *path*, as if with **open-input-file**, and passes this port to **procedure**. If **procedure** returns, **call-with-input-file** closes the input port and returns the values returned by **procedure**.

**call-with-input-file** does not automatically close the input port if a continuation created outside of **procedure** is invoked, since it is possible that another continuation created inside of **procedure** will be invoked at a later time, returning control to **procedure**. If **procedure** does not return, an implementation is free to close the input port only if it can prove that the input port is no longer accessible. As shown in Section 5.6, **dynamic-wind** may be used to ensure that the port is closed if a continuation created outside of **procedure** is invoked.

The following example shows the use of **call-with-input-file** in an expression that gathers a list of objects from the file named by "myfile.ss." It is functionally equivalent to the example given for **open-input-file** above.

```scheme
(call-with-input-file "myfile.ss"
  (lambda (p)
    (let f ([x (read p)])
      (if (eof-object? x)
        '(())
        (cons x (f (read p)))))
  ))
```

**call-with-input-file** might be defined without error checking as follows.

```scheme
(define call-with-input-file
  (lambda (filename proc)
    (let ([p (open-input-file filename)])
      (let-values ([v* (proc p)])
        (close-port p)
        (apply values v*)))))
```

218

Section 7.9. Convenience I/O
The Scheme Programming Language, 4th Edition

procedure: (call-with-output-file path procedure)
returns: the values returned by procedure
libraries: (rnrs io simple), (rnrs)

path must be a string or some other implementation-dependent value that names a file. procedure should accept one argument.

call-with-output-file creates a new output port for the file named by path, as if with open-output-file, and passes this port to procedure. If procedure returns, call-with-output-file closes the output port and returns the values returned by procedure.

call-with-output-file does not automatically close the output port if a continuation created outside of procedure is invoked, since it is possible that another continuation created inside of procedure will be invoked at a later time, returning control to procedure. If procedure does not return, an implementation is free to close the output port only if it can prove that the output port is no longer accessible. As shown in Section 5.6, dynamic-wind may be used to ensure that the port is closed if a continuation created outside of procedure is invoked.

The following shows the use of call-with-output-file to write a list of objects (the value of list-to-be-printed), separated by newlines, to the file named by "myfile.ss." It is functionally equivalent to the example given for open-output-file above.

(call-with-output-file "myfile.ss"
 (lambda (p)
   (let f ([ls list-to-be-printed])
     (unless (null? ls)
       (write (car ls) p)
       (newline p)
       (f (cdr ls))))))

call-with-output-file might be defined without error checking as follows.

(define call-with-output-file
 (lambda (filename proc)
   (let ([p (open-output-file filename)])
     (let-values ([v* (proc p)])
       (close-port p)
       (apply values v*)))))

procedure: (with-input-from-file path thunk)
returns: the values returned by thunk
libraries: (rnrs io simple), (rnrs)

path must be a string or some other implementation-dependent value that names a file. thunk must be a procedure and should accept zero arguments.

with-input-from-file temporarily changes the current input port to be the result of opening the file named by path, as if with open-input-file, during the application of thunk. If thunk returns, the port is closed and the current input port is restored to its old value.

The behavior of with-input-from-file is unspecified if a continuation created outside of thunk is invoked before thunk returns. An implementation may close the port and restore the current input port to its old value---but it may not.
procedure: (with-output-to-file path thunk)
returns: the values returned by thunk
libraries: (rnrs io simple), (rnrs)

path must be a string or some other implementation-dependent value that names a file. thunk must be a procedure and should accept zero arguments.

with-output-to-file temporarily rebinds the current output port to be the result of opening the file named by path, as if with open-output-file, during the application of thunk. If thunk returns, the port is closed and the current output port is restored to its old value.

The behavior of with-output-to-file is unspecified if a continuation created outside of thunk is invoked before thunk returns. An implementation may close the port and restore the current output port to its old value—but it may not.

procedure: (read)
procedure: (read textual-input-port)
returns: a Scheme datum object or the eof object
libraries: (rnrs io simple), (rnrs)

If textual-input-port is not supplied, it defaults to the current input port. This procedure is otherwise equivalent to get-datum.

procedure: (read-char)
procedure: (read-char textual-input-port)
returns: the next character from textual-input-port
libraries: (rnrs io simple), (rnrs)

If textual-input-port is not supplied, it defaults to the current input port. This procedure is otherwise equivalent to get-char.

procedure: (peek-char)
procedure: (peek-char textual-input-port)
returns: the next character from textual-input-port
libraries: (rnrs io simple), (rnrs)

If textual-input-port is not supplied, it defaults to the current input port. This procedure is otherwise equivalent to lookahead-char.

procedure: (write obj)
procedure: (write obj textual-output-port)
returns: unspecified
libraries: (rnrs io simple), (rnrs)

If textual-output-port is not supplied, it defaults to the current output port. This procedure is otherwise equivalent to put-datum, with the arguments reversed. See Section 12.5 for an implementation of put-datum, write, and display.

procedure: (display obj)
procedure: (display obj textual-output-port)
returns: unspecified
libraries: (rnrs io simple), (rnrs)

If textual-output-port is not supplied, it defaults to the current output port.
display is similar to write or put-datum but prints strings and characters found within obj directly. Strings are printed without quotation marks or escapes for special characters, as if by put-string, and characters are printed without the #\ notation, as if by put-char. With display, the three-element list (a b c) and the two-element list "a b" c) both print as (a b c). Because of this, display should not be used to print objects that are intended to be read with read. display is useful primarily for printing messages, with obj most often being a string. See Section 12.5 for an implementation of put-datum, write, and display.

procedure: (write-char char)
procedure: (write-char char textual-output-port)
returns: unspecified
libraries: (rnrs io simple), (rnrs)

If textual-output-port is not supplied, it defaults to the current output port. This procedure is otherwise equivalent to put-char, with the arguments reversed.

procedure: (newline)
procedure: (newline textual-output-port)
returns: unspecified
libraries: (rnrs io simple), (rnrs)

If textual-output-port is not supplied, it defaults to the current output port. newline sends a line-feed character to the port.

procedure: (close-input-port input-port)
procedure: (close-output-port output-port)
returns: unspecified
libraries: (rnrs io simple), (rnrs)

close-input-port closes an input port, and close-output-port closes an output port. These procedures are provided for backward compatibility with the Revised5 Report; they are not actually more convenient to use than close-port.

Section 7.10. Filesystem Operations

Scheme has two standard operations, beyond file input/output, for interacting with the filesystem: file-exists? and delete-file. Most implementations support additional operations.

procedure: (file-exists? path)
returns: #t if the file named by path exists, #f otherwise
libraries: (rnrs files), (rnrs)

path must be a string or some other implementation-dependent value that names a file. Whether file-exists? follows symbolic links is unspecified.

procedure: (delete-file path)
returns: unspecified
libraries: (rnrs files), (rnrs)

path must be a string or some other implementation-dependent value that names a file. delete-file removes the file named by path if it exists and can be deleted, otherwise it raises an exception with condition type &i/o-filename. Whether delete-file follows symbolic links is unspecified.
Section 7.11. Bytevector/String Conversions

The procedures described in this section encode or decode character sequences, converting from strings to bytevectors or bytevectors to strings. They do not necessarily involve input/output, though they might be implemented using bytevector input and output ports.

The first two procedures, `bytevector->string` and `string->bytevector`, take an explicit transcoder argument that determines the character encodings, eol styles, and error-handling modes. The others perform specific Unicode conversions with an implicit eol-style of `none` and error-handling mode of `replace`.

**procedure:** (bytevector->string bytevector transcoder)
**returns:** a string containing the characters encoded in bytevector
**libraries:** `(rnrs io ports), (rnrs)`

This operation, at least in effect, creates a bytevector input port with the specified `transcoder` from which all of the available characters are read, as if by `get-string-all`, and placed into the output string.

```
(let ([tx (make-transcoder (utf-8-codec) (eol-style lf) (error-handling-mode replace))]
    (bytevector->string #vu8(97 98 99) tx)) ⇒ "abc"
```

**procedure:** (string->bytevector string transcoder)
**returns:** a bytevector containing the encodings of the characters in string
**libraries:** `(rnrs io ports), (rnrs)`

This operation, at least in effect, creates a bytevector output port with the specified `transcoder` to which all of the characters of `string` are written, then extracts a bytevector containing the accumulated bytes.

```
(let ([tx (make-transcoder (utf-8-codec) (eol-style none) (error-handling-mode raise))]
    (string->bytevector "abc" tx)) ⇒ #vu8(97 98 99)
```

**procedure:** (string->utf8 string)
**returns:** a bytevector containing the UTF-8 encoding of string
**libraries:** `(rnrs bytevectors), (rnrs)`

**procedure:** (string->utf16 string)
**procedure:** (string->utf16 string endianness)
**procedure:** (string->utf32 string)
**procedure:** (string->utf32 string endianness)
**returns:** a bytevector containing the specified encoding of string
**libraries:** `(rnrs bytevectors), (rnrs)`

`endianness` must be one of the symbols `big` or `little`. If `endianness` is not provided or is the symbol `big`, `string->utf16` returns the UTF-16BE encoding of `string` and `string->utf32` returns the UTF-32BE encoding of `string`. If `endianness` is the symbol `little`, `string->utf16` returns the UTF-16LE encoding of `string` and `string->utf32` returns the UTF-32LE encoding of `string`. No byte-order mark is included in the encoding.

**procedure:** (utf8->string bytevector)
**returns:** a string containing the UTF-8 decoding of bytevector
**libraries:** `(rnrs bytevectors), (rnrs)`
procedure: (utf16->string bytevector endianness)
procedure: (utf16->string bytevector endianness endianness-mandatory?)
procedure: (utf32->string bytevector endianness)
procedure: (utf32->string bytevector endianness endianness-mandatory?)
returns: a string containing the specified decoding of bytevector
libraries: (rnrs bytevectors).

endianness must be one of the symbols big or little. These procedures return a UTF-16 or UTF-32 decoding of bytevector, with the endianness of the representation determined from the endianness argument or byte-order mark (BOM). If endianness-mandatory? is not provided or is #f, the endianness is determined by a BOM at the front of bytevector or, if no BOM is present, by endianness. If endianness-mandatory? is #t, the endianness is determined by endianness, and, if a BOM appears at the front of bytevector, it is treated as a regular character encoding.

The UTF-16 BOM is the two-byte sequence \( \text{x} \text{FE}, \text{x} \text{FF} \) specifying "big" or the two-byte sequence \( \text{x} \text{FF}, \text{x} \text{FE} \) specifying "little." The UTF-32 BOM is the four-byte sequence \( \text{x} \text{00}, \text{x} \text{00}, \text{x} \text{FE}, \text{x} \text{FF} \) specifying "big" or the four-byte sequence \( \text{x} \text{FF}, \text{x} \text{FE}, \text{x} \text{00}, \text{x} \text{00} \) specifying "little."
Chapter 8. Syntactic Extension

Syntactic extensions, or macros, are used to simplify and regularize repeated patterns in a program, to introduce syntactic forms with new evaluation rules, and to perform transformations that help make programs more efficient.

A syntactic extension most often takes the form \((\text{keyword} \ \text{subform} \ldots)\), where \text{keyword} is the identifier that names the syntactic extension. The syntax of each \text{subform} varies from one syntactic extension to another. Syntactic extensions can also take the form of improper lists or even singleton identifiers.

New syntactic extensions are defined by associating keywords with transformation procedures, or transformers. Syntactic extensions are defined using define-syntax forms or using let-syntax or letrec-syntax. Transformers may be created using syntax-rules, which allows simple pattern-based transformations to be performed. They may also be ordinary procedures that accept one argument and perform arbitrary computations. In this case, syntax-case is normally used to destruct the input and syntax is normally used to construct the output. The identifier-syntax form and make-variable-transformer procedure allow the creation of transformers that match singleton identifiers and assignments to those identifiers, the former being restricted to simple patterns like syntax-rules and the latter allowing arbitrary computations to be performed.

Syntactic extensions are expanded into core forms at the start of evaluation (before compilation or interpretation) by a syntax expander. If the expander encounters a syntactic extension, it invokes the associated transformer to expand the syntactic extension, then repeats the expansion process for the form returned by the transformer. If the expander encounters a core syntactic 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.

The syntactic extension mechanisms described in this chapter are part of the "syntax-case" system. A portable implementation of the system that also supports libraries and top-level programs is available at http://www.cs.indiana.edu/syntax-case/. A description of the motivations behind and implementation of the system can be found in the article "Syntactic Abstraction in Scheme" [12]. Additional features that have not yet been standardized, including modules, local import, and meta definitions, are described in the Chez Scheme User's Guide [9].

Section 8.1. Keyword Bindings

This section describes forms that establish bindings between keywords and transformers. Keyword bindings may be established within a top-level program or library body using define-syntax and in any local scope using define-syntax, let-syntax, or letrec-syntax.

\[
\text{syntax:} \quad \text{(define-syntax keyword expr)}
\]

\[
\text{libraries:} \quad \text{(rnrs base), (rnrs)}
\]

\text{expr} must evaluate to a transformer.

The following example defines \text{let*} as a syntactic extension, specifying the transformer with syntax-rules (see Section 8.2).

\[
\text{(define-syntax let*}
\text{ (syntax-rules ()
\quad [(_ () b1 b2 ...) (let () b1 b2 ...)]
\quad [(_ (((i1 e1) (i2 e2) ...) b1 b2 ...)
\quad (let ([i1 e1])
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
\text{)}
All bindings established by a set of internal definitions, whether keyword or variable definitions, are visible everywhere within the immediately enclosing body, including within the definitions themselves. For example, the expression

```
(let ()
  (define even?
    (lambda (x)
      (or (= x 0) (odd? (- x 1))))
  (define-syntax odd?
    (syntax-rules ()
      [(_ x) (not (even? x))])))
(even? 10))
```

is valid and should evaluate to #t.

The expander processes the initial forms in a library, lambda, or other body from left to right. If it encounters a variable definition, it 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. If it encounters a keyword definition, it expands and evaluates the right-hand-side expression and binds the keyword to the resulting transformer. If it encounters an expression, it fully expands all deferred right-hand-side expressions along with the current and remaining body expressions.

An implication of the left-to-right processing order 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 ()
      [(_ id) (define id 0)]))
  (bind-to-zero x)
)
```

evaluates to 0, regardless of any binding for bind-to-zero that might appear outside of the let expression.
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.

### Section 8.2. Syntax-Rules Transformers

The syntax-rules form described in this section permits simple transformers to be specified in a convenient manner. These transformers may be bound to keywords using the mechanisms described in Section 8.1. While it is much less expressive than the mechanism described in Section 8.3, it is sufficient for defining many common syntactic extensions.

**syntax:** (syntax-rules (literal ...) clause ...)

**returns:** a transformer

**libraries:** (rnrs base), (rnrs)

Each *literal* must be an identifier other than an underscore (_ _) or ellipsis ( ... ). Each clause must take the form below.

(pattern template)

Each *pattern* specifies one possible syntax that the input form might take, and the corresponding *template* specifies how the output should appear.

Patterns consist of list structure, vector structure, identifiers, and constants. Each identifier within a pattern is either a *literal*, a *pattern variable*, an underscore, or an *ellipsis*. The identifier _ is an underscore, and the identifier ... is an ellipsis. Any identifier other than _ or ... is a literal if it appears in the list of literals (literal ...); otherwise, it is a pattern variable. Literals serve as auxiliary keywords, such as else in case and cond expressions. List and vector structure within a pattern specifies the basic structure required of the input, the underscore and pattern variables specify arbitrary substructure, and literals and constants specify atomic pieces that must match exactly. Ellipses specify repeated occurrences of the subpatterns they follow.

An input form $F$ matches a pattern $P$ if and only if

- $P$ is an underscore or pattern variable,
- $P$ is a literal identifier and $F$ is an identifier with the same binding as determined by the predicate `free-identifier=?` (Section 8.3),
- $P$ 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 $n$th cdr matches $P_x$. 

Section 8.2. Syntax-Rules Transformers 227
The Scheme Programming Language, 4th Edition

- \( P \) is of the form \((P_1 \ldots P_k \ ellipsis P_{e+1} \ldots P_n)\), where \( \text{ellipsis} \) is the identifier \( \ldots \) 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 \( P_{e+1} \) and whose remaining \( n - m \) elements match \( P_{e+1} \) through \( P_n \).
- \( P \) is of the form \((P_1 \ldots P_k \ ellipsis P_{e+1} \ldots P_n . P_n)\), where \( \text{ellipsis} \) is the identifier \( \ldots \) 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 \( P_{e+1} \) whose next \( n - m \) elements match \( P_{e+1} \) through \( P_n \), and whose \( n \)th and final cdr matches \( P_n \).
- \( P \) is of the form \( \#(P_1 \ldots P_n) \) and \( F \) is a vector of \( n \) elements that match \( P_1 \) through \( P_n \).
- \( P \) is of the form \( \#(P_1 \ldots P_k \ ellipsis P_{e+1} \ldots P_n)\), where \( \text{ellipsis} \) is the identifier \( \ldots \) 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 \( P_{e+1} \), and whose remaining \( n - m \) elements match \( P_{e+1} \) through \( P_n \), or
- \( P \) is a pattern datum (any nonlist, nonvector, nonsymbol object) and \( F \) is equal to \( P \) in the sense of the equal? procedure.

The outermost structure of a syntax-rules pattern must actually be in one of the list-structured forms above, although subpatterns of the pattern may be in any of the above forms. Furthermore, the first element of the outermost pattern is ignored, since it is always assumed to be the keyword naming the syntactic form. (These statements do not apply to syntax-case; see Section 8.3.)

If an input form passed to a syntax-rules transformer matches the pattern for a given clause, the clause is accepted and the form is transformed as specified by the associated template. As this transformation takes place, pattern variables appearing in the pattern are bound to the corresponding input subforms. Pattern variables appearing within a subpattern followed by one or more ellipses may be bound to a sequence or sequences of zero or more input subforms.

A template is a pattern variable, an identifier that is not a pattern variable, a pattern datum, a list of subtemplates \((S_1 \ldots S_n)\), an improper list of subtemplates \((S_1 \ S_2 \ldots S_n \ . \ T)\), or a vector of subtemplates \(#(S_1 \ldots S_n)\). Each subtemplate \( S_i \) is a template followed by zero or more ellipses. The final element \( T \) of an improper subtemplate list is a template.

Pattern variables appearing within a template are replaced in the output by the input subforms to which they are bound. Pattern data and identifiers that are not pattern variables are inserted directly into the output. List and vector structure within the template remains list and vector structure in the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis. (Otherwise, the expander could not determine how many times the subform should be repeated in the output.) 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 template than in the associated pattern, the input form is replicated as necessary.

A template of the form \( \ldots \ template \) is identical to \( \text{template} \), except that ellipses within the template have no special meaning. That is, any ellipses contained within \( \text{template} \) are treated as ordinary identifiers. In particular, the template \( \ldots \ . \ . \ . \ ) \) produces a single ellipsis, \( \ldots \). This allows syntactic extensions to expand into forms containing ellipses, including \( \text{syntax-rules} \) or \( \text{syntax-case} \) patterns and templates.

The definition of or below demonstrates the use of syntax-rules.

\[
\text{(define-syntax or)}
\text{(syntax-rules ()}
\text{[(_ _ _ f)]}
\text{[(_ _ e) e]}
\]

228 Section 8.2. Syntax-Rules Transformers
The input patterns specify that the input must consist of the keyword and zero or more subexpressions. An underscore (\_), which is a special pattern symbol that matches any input, is often used for the keyword position to remind the programmer and anyone reading the definition that the keyword position never fails to contain the expected keyword and need not be matched. (In fact, as mentioned above, \texttt{syntax-rules} ignores what appears in the keyword position.) If more than one subexpression is present (third clause), the expanded code both tests the value of the first subexpression and returns the value if it is not false. To avoid evaluating the expression twice, the transformer introduces a binding for the temporary variable \texttt{t}.

The expansion algorithm maintains lexical scoping automatically by renaming local identifiers as necessary. Thus, the binding for \texttt{t} introduced by the transformer is visible only within code introduced by the transformer and not within subforms of the input. Similarly, the references to the identifiers \texttt{let} and \texttt{if} are unaffected by any bindings present in the context of the input.

\begin{verbatim}
(let ([t 'okay])
  (if t (begin e1 e2 ...)))
\end{verbatim}

This expression is transformed during expansion to the equivalent of the expression below.

\begin{verbatim}
((lambda (t1)
  ((lambda (t2)
    (if t2 t2 t1))
   'okay))
 #f) => okay
\end{verbatim}

In this sample expansion, \texttt{if1}, \texttt{t1}, and \texttt{t2} represent identifiers to which \texttt{if} and \texttt{t} in the original expression and \texttt{t} in the expansion of \texttt{or} have been renamed.

The definition of a simplified version of \texttt{cond} below (simplified because it requires at least one output expression per clause and does not support the auxiliary keyword \texttt{=>}) demonstrates how auxiliary keywords such as \texttt{else} are recognized in the input to a transformer, via inclusion in the list of literals.

\begin{verbatim}
(define-syntax cond
  (syntax-rules (else)
    [(_ (else e1 e2 ...)) (begin e1 e2 ...)]
    [(_ (e0 e1 e2 ...)) (if e0 (begin e1 e2 ...))]  
    [(_ (e0 e1 e2 ...) c1 c2 ...)
      (if e0 (begin e1 e2 ...) (cond c1 c2 ...))]])
\end{verbatim}
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 `tmpl`.

```
(let ()
  (define-syntax a (identifier-syntax car))
  (list (a '(1 2 3)) a)) ⇒ (1 #<procedure>)
```

With the first form of `identifier-syntax`, an apparent assignment of the associated keyword with `set!` is a syntax violation. The second, more general, form of `identifier-syntax` permits the transformer to specify what happens when `set!` is used.

```
(let ([ls (list 0)])
  (define-syntax a
    (identifier-syntax
      [id (car ls)]
      [(set! id e) (set-car! ls e)]))
  (let ([before a])
    (set! a 1)
    (list before a ls))) ⇒ (0 1 (1))
```

A definition of `identifier-syntax` in terms of `make-variable-transformer` is shown on page 307.

### Section 8.3. Syntax-Case Transformers

This section describes a more expressive mechanism for creating transformers, based on `syntax-case`, a generalized version of `syntax-rules`. This mechanism permits arbitrarily complex transformations to be specified, including transformations that "bend" lexical scoping in a controlled manner, allowing a much broader class of syntactic extensions to be defined. Any transformer that may be defined using `syntax-rules` may be rewritten easily to use `syntax-case` instead; in fact, `syntax-rules` itself may be defined as a syntactic extension in terms of `syntax-case`, as demonstrated within the description of `syntax-case` below.

With this mechanism, transformers are procedures of one argument. The argument is a `syntax object` representing the form to be processed. The return value is a syntax object representing the output form. A syntax object may be any of the following.

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

The `wrap` on a wrapped syntax object contains contextual information about a form in addition to its structure. This contextual information is used by the expander to maintain lexical scoping. The wrap may also contain information used by the implementation to correlate source and object code, e.g., track file, line, and character information through the expansion and compilation process.

The contextual information must be present for all identifiers, which is why the definition of syntax object above does not allow symbols unless they are wrapped. A 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,
Transformers normally destructure their input with `syntax-case` and rebuild their output with `syntax`. These two forms alone are sufficient for defining many syntactic extensions, including any that can be defined using `syntax-rules`. They are described below along with a set of additional forms and procedures that provide added functionality.

```
syntax: (syntax-case expr (literal ...) clause ...)
returns: see below
libraries: (rnrs syntax-case), (rnrs)
```

Each `literal` must be an identifier. Each `clause` must take one of the following two forms.

```
(pattern output-expression)
(pattern fender output-expression)
```

`syntax-case` patterns may be in any of the forms described in Section 8.2.

`syntax-case` first evaluates `expr`, then attempts to match the resulting value against the pattern from the first `clause`. This value may be any Scheme object. If the value matches the pattern and no `fender` is present, `output-expression` is evaluated and its values returned as the values of the `syntax-case` expression. If the value does not match the pattern, the value is compared against the next clause, 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 value of the `syntax-case expr` matches the pattern for a given clause, the corresponding `fender` is evaluated. If `fender` evaluates to a true value, the clause is accepted; otherwise, the clause is rejected as if the input had failed to match the pattern. Fenders are logically a part of the matching process, i.e., they specify additional matching constraints beyond the basic structure of an expression.

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 occupy the same namespace as program variables and keywords; pattern variable bindings created by `syntax-case` can shadow (and be shadowed by) program variable and keyword bindings as well as other pattern variable bindings. Pattern variables, however, can be referenced only within `syntax` expressions.

See the examples following the description of `syntax`.

```
syntax: (syntax template)
syntax: #'template
returns: see below
libraries: (rnrs syntax-case), (rnrs)
```

`#'template` is equivalent to `(syntax template)`. The abbreviated form is converted into the longer form when a program is read, prior to macro expansion.

A `syntax` expression is like a `quote` expression except that the values of pattern variables appearing within `template` are inserted into `template`, and contextual information associated both with the input and with the template is retained in the output to support lexical scoping. A `syntax template` is identical to a `syntax-rules template` and is treated similarly.

List and vector structures within the template become true lists or vectors (suitable for direct application of list or vector operations, like `map` or `vector-ref`) to the extent that the list or vector structures must be copied to insert the values of pattern variables, and empty lists are never wrapped. For example, `#'(x ...),`
#'(a b c), #'() are all lists if x, a, b, and c are pattern variables.

The definition of or below is equivalent to the one given in Section 8.2 except that it employs syntax-case and syntax in place of syntax-rules.

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

In this version, the lambda expression that produces the transformer is explicit, as are the syntax forms in the output part of each clause. Any syntax-rules form can be expressed with syntax-case by making the lambda expression and syntax expressions explicit. This observation leads to the following definition of syntax-rules in terms of syntax-case.

(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      [(_ (i ...) ((keyword . pattern) template) ...)
       #'(lambda (x)
           (syntax-case x (i ...)
             [(_ . pattern) #'template] ...))])))

An underscore is used in place of each keyword since the first position of each syntax-rules pattern is always ignored.

Since the lambda and syntax expressions are implicit in a syntax-rules form, definitions expressed with syntax-rules are often shorter than the equivalent definitions expressed with syntax-case. The choice of which to use when either suffices is a matter of taste, but many transformers that can be written easily with syntax-case cannot be written easily or at all with syntax-rules (see Section 8.4).

procedure: (identifier? obj)
returns: #t if obj is an identifier, #f otherwise
libraries: (rnrs syntax-case), (rnrs)

identifier? is often used within fenders to verify that certain subforms of an input form are identifiers, as in the definition of unnamed let below.

(define-syntax let
  (lambda (x)
    (define ids?
      (lambda (ls)
        (or (null? ls)
            (and (identifier? (car ls))
                 (ids? (cdr ls))))))
    (syntax-case x ()
      [(_ ((i e) ...) b1 b2 ...)
       (ids? #'(i ...))
       #'((lambda (i ...) b1 b2 ...) e ...)]))

Syntactic extensions ordinarily take the form (keyword subform ...), but the syntax-case system
permits them to take the form of singleton identifiers as well. For example, the keyword `pcar` in the expression below may be used both as an identifier (in which case it expands into a call to `car`) or as a structured form (in which case it expands into a call to `set-car!`).

\[
\begin{align*}
\text{(let ([p (cons 0 #f)] )} \\
\text{(define-syntax pcar} \\
\text{  (lambda (x) } \\
\text{     (syntax-case x ()} \\
\text{       [(_ (identifier? x) #'(car p)]} \\
\text{       [(_ e) #'(set-car! p e)]})} \\
\text{  )} \\
\text{(let ([a pcar] } \\
\text{    (pcar 1) } \\
\text{    (list a pcar))} \Rightarrow (0 1)
\end{align*}
\]

The fender `(identifier? x)` is used to recognize the singleton identifier case.

**procedure:** `(free-identifier=? identifier_1 identifier_2)`
**procedure:** `(bound-identifier=? identifier_1 identifier_2)`
**returns:** see below
**libraries:** `(rnrs syntax-case), (rnrs)`

Symbolic names alone do not distinguish identifiers unless the identifiers are to be used only as symbolic data. The predicates `free-identifier=?` and `bound-identifier=?` are used to compare identifiers according to their intended use as free references or bound identifiers in a given context.

`free-identifier=?` is used to determine whether two identifiers would be equivalent if they were to appear as free identifiers in the output of a transformer. Because identifier references are lexically scoped, this means `(free-identifier=? id_1 id_2)` is true if and only if the identifiers `id_1` and `id_2` refer to the same binding. (For this comparison, two like-named identifiers are assumed to have the same binding if neither is bound.) Literal identifiers (auxiliary keywords) appearing in `syntax-case` patterns (such as `else` in `case` and `cond`) are matched with `free-identifier=?`.

Similarly, `bound-identifier=?` is used to determine whether two identifiers would be equivalent if they were to appear as bound identifiers in the output of a transformer. In other words, if `bound-identifier=?` returns true for two identifiers, a binding for one will capture references to the other within its scope. 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`). `bound-identifier=?` 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.

The definition below is equivalent to the earlier definition of a simplified version of `cond` with `syntax-rules`, except that `else` is recognized via an explicit call to `free-identifier?` within a fender rather than via inclusion in the literals list.

\[
\begin{align*}
\text{(define-syntax cond} \\
\text{  (lambda (x) } \\
\text{     (syntax-case x ()} \\
\text{       [(_ (e0 e1 e2 ...))} \\
\text{         (and (identifier? #'e0) (free-identifier=? #'e0 #'else))} \\
\text{         #'(begin e1 e2 ...)]} \\
\text{       [(_ (e0 e1 e2 ...)) #'(if e0 (begin e1 e2 ...))]} \\
\text{       [(_ (e0 e1 e2 ...) c1 c2 ...)} \\
\text{         #'(if e0 (begin e1 e2 ...) (cond c1 c2 ...))}]})
\end{align*}
\]
With either definition of cond, else is not recognized as an auxiliary keyword if an enclosing lexical binding for else exists. For example,

```
(let ([else #f])
  (cond [else (write "oops")]))
```

does not write "oops", since else is bound lexically and is therefore not the same else that appears in the definition of cond.

The following definition of unnamed let uses bound-identifier=? to detect duplicate identifiers.

```
(define-syntax let
  (lambda (x)
    (define ids?
      (lambda (ls)
        (or (null? ls)
            (and (identifier? (car ls)) (ids? (cdr ls))))))
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (not (memp
                  (lambda (x) (bound-identifier=? x (car ls)))
                  (cdr ls))))
                (unique-ids? (cdr ls))))))
    (syntax-case x ()
      [(_ ((i e) ...) b1 b2 ...) (and (ids? #'(i ...)) (unique-ids? #'(i ...)))
        #'((lambda (i ...) b1 b2 ...) e ...))])))
```

With the definition of let above, the expression

```
(let ([a 3] [a 4]) (+ a a))
```

is a syntax violation, whereas

```
(let ([a 0])
  (let-syntax ([dolet (lambda (x)
                        (syntax-case x ()
                          [(_ b)
                            #'(let ([a 3] [b 4]) (+ a b)))]))
    (dolet a)))
```

evaluates to 7 since the identifier a introduced by dolet and the identifier a extracted from the input form are not bound-identifier=?.

Since both occurrences of a, however, if left as free references, would refer to the same binding for a, free-identifier=? would not distinguish them.

Two identifiers that are free-identifier=? may not be bound-identifier=?.

An identifier introduced by a transformer may refer to the same enclosing binding as an identifier not introduced by the transformer, but an introduced binding for one will not capture references to the other. On the other hand, identifiers that are bound-identifier=? are free-identifier=?, as long as the identifiers have valid bindings in the context where they are compared.

**syntax**: (with-syntax (((pattern expr) ...) body₁ body₂ ...))

**returns**: the values of the final body expression
It is sometimes useful to construct a transformer’s output in separate pieces, then put the pieces together. with-syntax facilitates this by allowing the creation of local pattern bindings.

pattern is identical in form to a syntax-case pattern. The value of each expr is computed and destructured according to the corresponding pattern, and pattern variables within the pattern are bound as with syntax-case to appropriate portions of the value within the body body₁ body₂ ..., which is processed and evaluated like a lambda body.

with-syntax may be defined as a syntactic extension in terms of syntax-case.

(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      [((_ ((p e) ...) b1 b2 ...) #'(syntax-case (list e ...) ()
        [(p ...) (let () b1 b2 ...)])])
    (define-syntax cond
      (lambda (x)
        (syntax-case x ()
          [((_ c1 c2 ...) (let f ([c1 #'c1] [cmore #'(c2 ...)])
            (if (null? cmore)
              (syntax-case c1 (else =
                [(else e1 e2 ...) #'(begin e1 e2 ...)])
              [(e0) #'(let () (if t (e1 t) (e1 t)))
                [(e0 => e1) #'(let () (if t (e1 t) rest))
                  [(e0 e1 e2 ...) #'(if e0 (begin e1 e2 ...) rest)])])
            (with-syntax ([rest (f (car cmore) (cdr cmore))]
              (syntax-case c1 (=
                [(e0) #'(let () (if t t) rest))
                [(e0 => e1) #'(let () (if t (e1 t) rest))
                  [(e0 e1 e2 ...) #'(if e0 (begin e1 e2 ...) rest)])]))))))))

syntax: (quasisyntax template ...
syntax: #\template
syntax: (unsyntax template ...
syntax: #,template
syntax: (unsyntax-splicing template ...
syntax: #,@template
returns: see below
libraries: (rnrs syntax-case), (rnrs)
similar to quasiquote (Section 6.1).

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. unsyntax and unsyntax-splicing are valid only within quasisyntax expressions.

quasisyntax expressions 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.

quasisyntax can be used in place of with-syntax in many cases. For example, the following definition of case employs quasisyntax to construct its output, using internal recursion in a manner similar to the definition of cond given under the description of with-syntax above.

```scheme
(define-syntax case
  (lambda (x)
    (syntax-case x ()
      [(_) e cl c2 ...]
      #'(let ([t e])
          (begin unsyntax cl (else)
            [(else el e2 ...)]
            (begin unsyntax el e2 ...)
            [(k ...) el e2 ...]
            (begin unsyntax (k ...))
            (begin unsyntax cl (k ...))
            `(if (memv t '(k ...)) (begin e1 e2 ...))
            #,(f (car cmore) (cdr cmore)))))))
```

unsyntax and unsyntax-splicing forms that contain 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. They support certain useful nested quasiquotation (quasisyntax) idioms [3], 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 6.1.

unsyntax and unsyntax-splicing are auxiliary keywords for quasisyntax. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

**procedure**: (make-variable-transformer procedure)

**returns**: a variable transformer

**libraries**: (rnrs syntax-case), (rnrs)

As described in the lead-in to this section, transformers may simply be procedures that accept one argument, a syntax object representing the input form, and return a new syntax object representing the output form. The form passed to a transformer usually represents a parenthesized form whose first subform is the keyword bound to the transformer or just the keyword itself. make-variable-transformer may be used to
convert a procedure into a special kind of transformer to which the expander also passes `set!` forms in which the keyword appears just after the `set!` keyword, as if it were a variable to be assigned. This allows the programmer to control what happens when the keyword appears in such contexts. The argument, `procedure`, should accept one argument.

```
(let ([ls (list 0)])
  (define-syntax a
    (make-variable-transformer
     (lambda (x)
       (syntax-case x ()
         [id (identifier? #id) #'(car ls)]
         [(set! _ e) #'(set-car! ls e)]
         [(_ e ...) #'((car ls) e ...)]))))
  (let ([before a])
    (set! a 1)
    (list before a ls))) ⇒ (0 1 (1))
```

This syntactic abstraction can be defined more succinctly using `identifier-syntax`, as shown in Section 8.2, but `make-variable-transformer` can be used to create transformers that perform arbitrary computations, while `identifier-syntax` is limited to simple term rewriting, like `syntax-rules`. `identifier-syntax` can be defined in terms of `make-variable-transformer`, as shown below.

```
(define-syntax identifier-syntax
  (lambda (x)
    (syntax-case x (set!)
      [(_ e)
       #'(lambda (x)
           (syntax-case x ()
             [id (identifier? #id) #'e]
             [(_ x (... ...)) #'(e x (... ...))])))
      [(_ (id exp1) ((set! var val) exp2))
       #'(make-variable-transformer
           (lambda (x)
             (syntax-case x (set!)
               [(set! var val) #'exp2]
               [(id x (... ...)) #'(exp1 x (... ...))]])
               [id (identifier? #id) #'exp1]))]))
```

**procedure**: `(syntax->datum obj)  
returns: `obj` stripped of syntactic information  
libraries: `(rnrs syntax-case), (rnrs)`

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))))
```
Two identifiers that are free-identifier=? need not be symbolic-identifier=?: two identifiers that refer to the same binding usually have the same name, but the rename and prefix subforms of the library’s import form (page 243) may result in two identifiers with different names but the same binding.

**procedure**: (datum->syntax template-identifier obj)

**returns**: a syntax object

**libraries**: (rnrs syntax-case), (rnrs)

`datum->syntax` constructs a syntax object from `obj` that contains the same contextual information as `template-identifier`, with the effect that the syntax object behaves as if it were introduced into the code when `template-identifier` was introduced. The template identifier is often the keyword of an input form, extracted from the form, and the object is often a symbol naming an identifier to be constructed.

`datum->syntax` 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 syntactic extensions that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form. For example, we can define a `loop` expression that binds the variable `break` to an escape procedure within the loop body.

```scheme
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(k e ...)
       (with-syntax ([break (datum->syntax #'k 'break)])
        #'(call/cc
            (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 we to define `loop` as

```scheme
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(_ e ...)
       #'(call/cc
            (lambda (break)
              (let f () e ... (f))))]))
  
  the variable `break` would not be visible in `e` ....
```

It is also useful for `obj` to represent an arbitrary Scheme form, as demonstrated by the following definition of `include`.

```scheme
(define-syntax include
  (lambda (x)
    (define read-file
      (lambda (fn k)
        (let ([p (open-input-file fn)])
          (let ((
```
(let f ([x (read p)])
  (if (eof-object? x)
      (begin (close-port p) '())
      (cons (datum->syntax k x) (f (read p)))))))

(syntax-case x ()
  [(k filename)
    (let ([fn (syntax->datum #'filename)])
      (with-syntax ([expr ...] (read-file fn #'k))
        #'(begin expr ...)))]))

(include "filename") expands into a `begin` expression containing the forms found in the file named by "filename". For example, if the file `f-def.ss` contains the expression

(define f (lambda () x))

(f)

the expression

(let ([x "okay"])
  (include "f-def.ss")
  (f))

evaluates to "okay".

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.

procedure: `(generate-temporaries list)`
returns: a list of distinct generated identifiers
libraries: `(rnrs syntax-case), (rnrs)`

Transformers can introduce a fixed number of identifiers into their output 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.

`list` may be any list; its contents are not important. The number of temporaries generated is the number of elements in `list`. Each temporary is guaranteed to be 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 e) ...) b1 b2 ...) (with-syntax ([expr ...] (read-file fn #'k))
          #'(let ([i #f] ...)
              (let ([t e] ...)
                (set! i t)
                ... (let () b1 b2 ...)))]))))

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 temporary identifiers have been generated.

Here is a definition of `let-values` (page 99) that uses this technique to support multiple sets of bindings.
The implementation of lvhelp is complicated by the need to evaluate all of the right-hand-side expressions before creating any of the bindings and by the need to support improper formals lists.

**Section 8.4. Examples**

This section presents a series of illustrative syntactic extensions defined with either `syntax-rules` or `syntax-case`, starting with a few simple but useful syntactic extensions and ending with a fairly complex mechanism for defining structures with automatically generated constructors, predicates, field accessors, and field setters.

The simplest example in this section is the following definition of `rec`. `rec` is a syntactic extension that permits internally recursive anonymous (not externally named) procedures to be created with minimal effort.

```
(define-syntax rec
  (syntax-rules ()
    [(_ x e) (letrec ([x e]) x)]))

(map rec sum
  (lambda (x)
    (if (= x 0)
        0
        (+ x (sum (- x 1))))))

'(0 1 2 3 4 5) ⇒ (0 1 3 6 10 15)
```

Using `rec`, we can define the full `let` (both unnamed and named) as follows.

```
(define-syntax let
  (syntax-rules ()
    [(_ ((x e) ...) b1 b2 ...) ((lambda (x ...) b1 b2 ...) e ...)]
    [(_ f ((x e) ...) b1 b2 ...) ((rec f (lambda (x ...) b1 b2 ...)) e ...)])
```
We can also define let directly in terms of letrec, although the definition is a bit less clear.

```
(define-syntax let
  (syntax-rules ()
    [(_ ((x e) ...) b1 b2 ...)
      ((lambda (x ...) b1 b2 ...) e ...)]
    [(_ f ((x e) ...) b1 b2 ...)
      ((letrec ([f (lambda (x ...) b1 b2 ...)]) f) e ...)]))
```

These definitions rely upon the fact that the first pattern cannot match a named let, since the first subform of a named let must be an identifier, not a list of bindings. The following definition uses a fender to make this check more robust.

```
(define-syntax let
  (lambda (x)
    (syntax-case x ()
      [(_ ((x e) ...) b1 b2 ...)
        #'((lambda (x ...) b1 b2 ...) e ...)]
      [(_ f ((x e) ...) b1 b2 ...)
        (identifier? #'f)
        #'((rec f (lambda (x ...) b1 b2 ...)) e ...)])))
```

With the fender, we can even put the clauses in the opposite order.

```
(define-syntax let
  (lambda (x)
    (syntax-case x ()
      [(_ f ((x e) ...) b1 b2 ...)
        (identifier? #'f)
        #'((rec f (lambda (x ...) b1 b2 ...)) e ...)]
      [(_ ((x e) ...) b1 b2 ...)
        #'((lambda (x ...) b1 b2 ...) e ...)])))
```

To be completely robust, the ids? and unique-ids? checks employed in the definition of unnamed let in Section 8.3 should be employed here as well.

Both variants of let are easily described by simple one-line patterns, but do requires a bit more work. The precise syntax of do cannot be expressed directly with a single pattern because some of the bindings in a do expression's binding list may take the form (var val) while others take the form (var val update). The following definition of do uses syntax-case internally to parse the bindings separately from the overall form.

```
(define-syntax do
  (lambda (x)
    (syntax-case x ()
      [(_ (binding ...) (test res ...) expr ...) (with-syntax ([((var val update) ...)]
        (map (lambda (b)
          (syntax-case b ()
            [(var val) #'(var val var)]
            [(var val update) #'(var val update)]))
          #'(binding ...)]))
        #'(let doloop ([var val] ...) (if test
```
The odd-looking expression \((\text{if} \ \#f \ \#f)\) is inserted before the result expressions \(\text{res} \ldots\) in case no result expressions are provided, since \(\text{begin}\) requires at least one subexpression. The value of \((\text{if} \ \#f \ \#f)\) is unspecified, which is what we want since the value of \(\text{do}\) is unspecified if no result expressions are provided. At the expense of a bit more code, we could use \(\text{syntax-case}\) to determine whether any result expressions are provided and to produce a loop with either a one- or two-armed \(\text{if}\) as appropriate. The resulting expansion would be cleaner but semantically equivalent.

As mentioned in Section 8.2, ellipses lose their special meaning within templates of the form \((\ldots \text{template})\). This fact allows syntactic extensions to expand into syntax definitions containing ellipses. This usage is illustrated by the definition below of \(\text{be-like-begin}\).

\[
\text{(define-syntax be-like-begin}
  \ (\text{syntax-rules} ()
    \ [(_ \text{name})
    \ (\text{define-syntax} \text{name}
    \ (\text{syntax-rules} ()
      \ [(_ \text{e0} \text{e1} \ldots)
        \ (\text{begin} \text{e0} \text{e1} \ldots))]
    )))
\]

With \(\text{be-like-begin}\) defined in this manner, \((\text{be-like-begin} \text{sequence})\) has the same effect as the following definition of \(\text{sequence}\).

\[
\text{(define-syntax sequence}
  \ (\text{syntax-rules} ()
    \ [(_ \text{e0} \text{e1} \ldots)
      \ (\text{begin} \text{e0} \text{e1} \ldots))]
\]

That is, a \(\text{sequence}\) form becomes equivalent to a \(\text{begin}\) form so that, for example:

\[
\text{(sequence} \ (\text{display} \ "\text{Say what?}\)\) \ (\text{newline}))
\]

prints "Say what?" followed by a newline.

The following example shows how one might restrict \(\text{if}\) expressions within a given expression to require the "else" (alternative) subexpression by defining a local \(\text{if}\) in terms of the built-in \(\text{if}\). Within the body of the \(\text{let-syntax}\) binding below, two-armed \(\text{if}\) works as always:

\[
\text{(let-syntax} \ (\text{[if} \ (\text{lambda} \ (x)}
  \ (\text{syntax-case} \ x ()
    \ [(_ \text{e1} \text{e2} \text{e3})
      \ "!(\text{if} \text{e1} \text{e2} \text{e3})])])
\]
\]

\[
(\text{if} \ (< \ 1 \ 5) \ 2 \ 3) \Rightarrow 2
\]

but one-armed \(\text{if}\) results in a syntax error.

\[
\text{(let-syntax} \ (\text{[if} \ (\text{lambda} \ (x)}
  \ (\text{syntax-case} \ x ()
    \ [(_ \text{e1} \text{e2} \text{e3})
      \ "!(\text{if} \text{e1} \text{e2} \text{e3})])])
\]
\]

\[
(\text{if} \ (< \ 1 \ 5) \ 2) \Rightarrow \text{syntax violation}
\]

Although this local definition of \(\text{if}\) looks simple enough, there are a few subtle ways in which an attempt to
write it might go wrong. If letrec-syntax were used in place of let-syntax, the identifier if inserted into the output would refer to the local if rather than the built-in if, and expansion would loop indefinitely.

Similarly, if the underscore were replaced with the identifier if, expansion would again loop indefinitely. The if appearing in the template (if e1 e2 e3) would be treated as a pattern variable bound to the corresponding identifier if from the input form, which denotes the local version of if.

Placing if in the list of literals in an attempt to patch up the latter version would not work either. This would cause syntax-case to compare the literal if in the pattern, which would be scoped outside the let-syntax expression, with the if in the input expression, which would be scoped inside the let-syntax. Since they would not refer to the same binding, they would not be free-identifier=?, and a syntax violation would result.

The conventional use of underscore (_) helps the programmer avoid situations like these in which the wrong identifier is matched against or inserted by accident.

It is a syntax violation to generate a reference to an identifier that is not present within the context of an input form, which can happen if the "closest enclosing lexical binding" for an identifier inserted into the output of a transformer does not also enclose the input form. For example,

```scheme
(let-syntax ((divide (lambda (x)
                          (let ([/ +])
                            (syntax-case x ()
                                [([el e2) #'(/ e1 e2)])])))
              ([/ *]) (divide 2 1)))
```

should result in a syntax violation with a message to the effect that / is referenced in an invalid context, since the occurrence of / in the output of divide is a reference to the variable / bound by the let expression within the transformer.

The next example defines a define-integrable form that is similar to define for procedure definitions except that it causes the code for the procedure to be integrated, or inserted, wherever a direct call to the procedure is found.

```scheme
(define-syntax define-integrable
  (syntax-rules (lambda)
                  [(_ name (lambda formals form1 form2 ...))
                   (begin
                     (define xname (lambda formals form1 form2 ...))
                     (define-syntax name
                       (lambda (x)
                         (syntax-case x ()
                             [(_ (identifier? x) #'xname]
                             [(_ arg (... ...))
                              #'((lambda formals form1 form2 ...) arg
                                (... ...)))])]]))
```

The form (define-integrable name lambda-expression) expands into a pair of definitions: a syntax definition of name and a variable definition of xname. The transformer for name converts apparent calls to name into direct calls to lambda-expression. Since the resulting forms are merely direct lambda applications (the equivalent of let expressions), the actual parameters are evaluated exactly once and before evaluation of the procedure's body, as required. All other references to name are replaced with references to xname. The definition of xname binds it to the value of lambda-expression. This allows
the procedure to be used as a first-class value. The define-integrable transformer does nothing special to maintain lexical scoping within the lambda expression or at the call site, since lexical scoping is maintained automatically by the expander. Also, because xname is introduced by the transformer, the binding for xname is not visible anywhere except where references to it are introduced by the the transformer for name.

The above definition of define-integrable does not work for recursive procedures, since a recursive call would cause an indefinite number of expansion steps, likely resulting in exhaustion of memory at expansion time. A solution to this problem for directly recursive procedures is to wrap each occurrence of the lambda expression with a let-syntax binding that unconditionally expands name to xname.

```
(define-syntax define-integrable
  (syntax-rules (lambda)
    [(_ name (lambda formals form1 form2 ...))
      (begin
        (define xname
          (let-syntax ([name (identifier-syntax xname)])
            (lambda formals form1 form2 ...)))
        (define-syntax name
          (lambda (x)
            (syntax-case x ()
              [(_ (identifier? x) #'xname]
                [(_ arg (... ...))
                  #'((let-syntax ([name (identifier-syntax xname)])
                      (lambda formals form1 form2 ...))
                    arg (... ...))))))))
    ])
)
```

This problem can be solved for mutually recursive procedures by replacing the let-syntax forms with the nonstandard fluid-let-syntax form, which is described in the Chez Scheme User's Guide [9].

Both definitions of define-integrable treat the case where an identifier appears in the first position of a structured expression differently from the case where it appears elsewhere, as does the pcar example given in the description for identifier?. In other situations, both cases must be treated the same. The form identifier-syntax can make doing so more convenient.

```
(let ([x 0])
  (define-syntax x++
    (identifier-syntax
      (let ([t x])
        (set! x (+ t 1)) t)))
  (let ([a x++] (list a x)]) => (0 1)
```

The following example uses identifier-syntax, datum->syntax, and local syntax definitions to define a form of method, one of the basic building blocks of object-oriented programming (OOP) systems. A method expression is similar to a lambda expression, except that in addition to the formal parameters and body, a method expression also contains a list of instance variables (ivar ...). When a method is invoked, it is always passed an object (instance), represented as a vector of fields corresponding to the instance variables, and zero or more additional arguments. Within the method body, the object is bound implicitly to the identifier self and the additional arguments are bound to the formal parameters. The fields of the object may be accessed or altered within the method body via instance variable references or assignments.

```
(define-syntax method
  (lambda (x)
```

Section 8.4. Examples
Local bindings for `ivar ...` and for `set!` make the fields of the object appear to be ordinary variables, with references and assignments translated into calls to `vector-ref` and `vector-set!`. `datum->syntax` is used to make the introduced bindings of `self` and `set!` visible in the method body. Nested `let-syntax` expressions are needed so that the identifiers `ivar ...` serving as auxiliary keywords for the local version of `set!` are scoped properly.

By using the general form of `identifier-syntax` to handle `set!` forms more directly, we can simplify the definition of `method`.

```scheme
(define-syntax method
  (lambda (x)
    (syntax-case x ()
      [(k (ivar ...) formals b1 b2 ...)]
        (with-syntax ([(index ...)]
                      (let f ([i 0] [ls #'(ivar ...)])
                          (if (null? ls)
                              ()
                              (cons i (f (+ i 1) (cdr ls)))))
                        [self (datum->syntax #'k 'self)]
                        [set! (datum->syntax #'k 'set!)])
                    #'(lambda (self . formals)
                       (let-syntax ([(ivar (identifier-syntax
                                          [_ (vector-ref self index)])
                                   ...
                                   (set! _ e)
                                   (vector-set! self index e))])))
                         ...)
                         (let-syntax ([(set! rules (ivar ...)
                                        (vector-set! self index e)]
                                        ...
                                        (set! x e)))
                             b1 b2 ...))))))))
```

The examples below demonstrate simple uses of `method`.

```scheme
(let ([m (method (a) (x) (list a x self))])
  (m #(1) 2)) ⇒ (1 2 #(1))

(let ([m (method (a) (x)
           (set! a x)
           (vector-set! self index e))]
     ...))
```

Section 8.4. Examples
In a complete OOP system based on method, the instance variables `ivar ...` would likely be drawn from class declarations, not listed explicitly in the method forms, although the same techniques would be used to make instance variables appear as ordinary variables within method bodies.

The final example of this section defines a simple structure definition facility that represents structures as vectors with named fields. Structures are defined with `define-structure`, which takes the form:

```
(define-structure name field ...)
```

where `name` names the structure and `field ...` names its fields. `define-structure` expands into a series of generated definitions: a constructor `make-name`, a type predicate `name?`, and one accessor `name-field` and setter `set-name-field!` per field name.

```scheme
(define-syntax define-structure
  (lambda (x)
    (define gen-id
      (lambda (template-id . args)
        (datum->syntax template-id
          (string->symbol
            (apply string-append
              (map (lambda (x)
                      (if (string? x)
                          x
                          (symbol->string (syntax->datum x))))
              args))))))
    (syntax-case x ()
      [(_ name field ...)
       (with-syntax ([constructor (gen-id #'name "make-" #'name)]
                     [predicate (gen-id #'name #'name ":")]
                     [(access ...) (map (lambda (x) (gen-id x #'name "-" x))
                                   #'(field ...))]
                     [(assign ...) (map (lambda (x)
                                          (gen-id x "set-" #'name "-" x ":")
                                          #'(field ...))]
                     [structure-length (+ (length #'(field ...)) 1)]
                     [(index ...) (let f ([i 1] [ids #'(field ...)])
                                   (if (null? ids)
                                       ()
                                       (cons i (f (+ i 1) (cdr ids)))))
                       '#'(begin
                           (define constructor
                             (lambda (field ...)
                               (vector 'name field ...)))
                           (define predicate
                             (lambda (x)
                               (and (vector? x)
                                    (= (vector-length x) structure-length))
```
The constructor accepts as many arguments as there are fields in the structure and creates a vector whose first element is the symbol `name` and whose remaining elements are the argument values. The type predicate returns true if its argument is a vector of the expected length whose first element is `name`.

Since a `define-structure` form expands into a `begin` containing definitions, it is itself a definition and can be used wherever definitions are valid.

The generated identifiers are created with `datum->syntax` to allow the identifiers to be visible where the `define-structure` form appears.

The examples below demonstrate the use of `define-structure`.

```scheme
(define-structure tree left right)
(define t
  (make-tree
    (make-tree 0 1)
    (make-tree 2 3)))

t => #(tree #(tree 0 1) #(tree 2 3))
(tree? t) => #t
(tree-left t) => #(tree 0 1)
(tree-right t) => #(tree 2 3)
(set-tree-left! t 0)
t => #(tree 0 #(tree 2 3))
```

R. Kent Dybvig / The Scheme Programming Language, Fourth Edition
Copyright © 2009 The MIT Press. Electronically reproduced by permission.
Illustrations © 2009 Jean-Pierre Hébert
ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
to order this book / about this book

http://www.scheme.com

Section 8.4. Examples
Section 8.4. Examples
Chapter 9. Records

This chapter describes the means by which the programmer may define new data types, or records types, each distinct from all other types. A record type determines the number and names of the fields each instance of the type has. Records are defined via the define-record-type form or the make-record-type-descriptor procedure.

Section 9.1. Defining Records

A define-record-type form defines a record type and, along with it, a constructor procedure for records of the type, a type predicate that returns true only for records of the type, an access procedure for each field, and an assignment procedure for each mutable field. For example, the definition

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

creates a point record type with two fields, x and y, and defines the following procedures:

```
(make-point x y)    constructor
(point? obj)        predicate
(point-x p)         accessor for field x
(point-y p)         accessor for field y
```

With this definition in place, we can use these procedures to create and manipulate records of the point type, as illustrated below.

```
(define p (make-point 36 -17))
(point? p) ⇒ #t
(point? '(cons 36 -17)) ⇒ #f
(point-x p) ⇒ 36
(point-y p) ⇒ -17
```

Fields are immutable by default, but may be declared mutable. In the alternate definition of point below, the x field is mutable while y remains immutable.

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

In this case, define-record-type defines a mutator for the x field in addition to the other products shown above.

```
(point-x-set! p x)   mutator for field x
```

The mutator can be used to change the contents of the x field.

```
(define p (make-point 36 -17))
(point-x-set! p (- (point-x p) 12))
(point-x p) ⇒ 24
```

A field may be declared immutable explicitly for clarity; the definition of point below is equivalent to the second definition above.

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

The names of the procedures defined by define-record-type follow the regular naming convention illustrated by the examples above, by default, but the programmer can override the defaults if desired. With
the following definition of \texttt{point}, the constructor is \texttt{mkpoint}, the predicate is \texttt{ispoint?}, and the
accessors for \texttt{x} and \texttt{y} are \texttt{x-val} and \texttt{y-val}. The mutator for \texttt{x} is \texttt{set-x-val!}.

\begin{verbatim}
(define-record-type (point mkpoint ispoint?)
  (fields (mutable x x-val set-x-val!)
           (immutable y y-val)))
\end{verbatim}

By default, a record definition creates a new type each time it is evaluated, as illustrated by the example
below.

\begin{verbatim}
(define (f p)
  (define-record-type point (fields x y))
  (if (eq? p 'make) (make-point 3 4) (point? p)))
(f (f 'make)) ⇒ #f
\end{verbatim}

The first (inner) call to \texttt{f} returns a point \texttt{p}, which is passed to \texttt{f} in the second (outer) call, which applies
\texttt{point?} to \texttt{p}. This \texttt{point?} is looking for points of the type created by the second call, while \texttt{p} is a point of
the type created by the first call. So \texttt{point?} returns \texttt{#f}.

This default \textit{generative} behavior may be overridden by including a \texttt{nongenerative} clause in the record
definition.

\begin{verbatim}
(define (f p)
  (define-record-type point (fields x y) (nongenerative))
  (if (eq? p 'make) (make-point 3 4) (point? p)))
(define p (f 'make))
(f p) ⇒ #t
\end{verbatim}

Record types created in this manner are still distinct from record types created by a definition appearing in a
different part of the program, even if the definitions are syntactically identical:

\begin{verbatim}
(define (f)
  (define-record-type point (fields x y) (nongenerative))
  (make-point 3 4))
(define (g p)
  (define-record-type point (fields x y) (nongenerative))
  (point? p))
(g (f)) ⇒ #f
\end{verbatim}

Even this can be overridden by including a uid (unique id) in the \texttt{nongenerative} clause:

\begin{verbatim}
(define (f)
  (define-record-type point (fields x y) (nongenerative really-the-same-point))
  (make-point 3 4))
(define (g p)
  (define-record-type point (fields x y) (nongenerative really-the-same-point))
  (point? p))
(g (f)) ⇒ #t
\end{verbatim}

The uid may be any identifier, but programmers are encouraged to select uids from the RFC 4122 UUID
namespace [20], possibly with the record-type name as a prefix.
A record type may be defined as a subtype of an existing "parent" type with a parent clause that declares the name of the existing record type. If a parent is specified, the new "child" record type inherits the parent record's fields, and each instance of the child type is considered to be an instance of the parent type, so that accessors and mutators for the parent type may be used on instances of the child type.

```
(define-record-type point (fields x y))
(define-record-type cpoint (parent point) (fields color))
```

The child type has all of the fields of the parent type, plus the additional fields declared in the child's definition. This is reflected in the constructor for cpoint, which now takes three arguments, with the parent arguments followed by the child argument.

```
(define cp (make-cpoint 3 4 'red))
```

A record of the child type is considered a record of the parent type, but a record of the parent type is not a record of the new type.

```
(point? (make-cpoint 3 4 'red)) ⇒ #t
(cpoint? (make-point 3 4)) ⇒ #f
```

Only one new accessor is created for cpoint, the one for the new field color. The existing accessors and mutators for the parent type may be used to access and modify the parent fields of the child type.

```
(define cp (make-cpoint 3 4 'red))
(point-x cp) ⇒ 3
(point-y cp) ⇒ 4
(cpoint-color cp) ⇒ red
```

As the examples given so far illustrate, the default constructor defined by define-record-type accepts as many arguments as the record has fields, including parent fields, and parent's parent fields, and so on. The programmer may override the default and specify the arguments to the constructor for the new type and how it determines the initial values of the constructed record's fields, via the protocol clause. The following definition creates a point record with three fields: x, y, and d, where d represents the displacement from the origin. The constructor still takes only two arguments, the x and y values, and initializes d to the square root of the sum of the squares of x and y.

```
(define-record-type point
  (fields x y d)
  (protocol
    (lambda (new)
      (lambda (x y)
        (new x y (sqrt (+ (* x x) (* y y)))))))))
```

```
(define p (make-point 3 4))
(point-x p) ⇒ 3
(point-y p) ⇒ 4
(point-d p) ⇒ 5
```

The procedure value of the expression within the protocol clause receives as an argument a primitive constructor new and returns a final constructor c. There are essentially no limits on what c is allowed to do, but if it returns, it should return the result of calling new. Before it does so, it may modify the new record instance (if the record type has mutable fields), register it with some external handler, print messages, etc. In this case, c accepts two arguments, x and y, and applies new to x, y, and the result of computing the origin displacement based on x and y.
If a parent record is specified, the construction protocol becomes more involved. The following definition of `cpoint` assumes that `point` has been defined as shown just above.

```scheme
(define-record-type cpoint
  (parent point)
  (fields color)
  (protocol
    (lambda (pargs->new)
      (lambda (c x y)
        ((pargs->new x y) c))))))
```

```scheme
(define cp (make-cpoint 'red 3 4))
(point-x cp) ⇒ 3
(point-y cp) ⇒ 4
(point-d cp) ⇒ 5
(cpoint-color cp) ⇒ red
```

Because a parent clause is present, the procedure value of the expression within the `protocol` clause receives a procedure `pargs->new` that, when applied to parent arguments, returns a new procedure. The new procedure, when passed the values of the child fields, returns the result of applying the parent protocol to an appropriate new procedure of its own. In this case, `pargs->new` is passed the values of the child constructor's second and third arguments (the `x` and `y` values) and the resulting new procedure is passed the value of the child constructor's first argument (the color). Thus, the protocol supplied in this example effectively reverses the normal order of arguments in which the parent arguments come before the child arguments, while arranging to pass along the arguments needed by the parent protocol.

The default protocol is equivalent to

```scheme
(lambda (new) new)
```

for record types with no parents, while for record types with parents, the default protocol is equivalent to the following

```scheme
(lambda (pargs->new)
  (lambda (x₁ ... xₙ y₁ ... yₘ)
    ((pargs->new x₁ ... xₙ) y₁ ... yₘ)))
```

where `n` is the number of parent (including grandparent, etc.) fields and `m` is the number of child fields.

Use of the `protocol` clause insulates the child record definition from some changes to the parent record type. The parent definition may be modified to add or remove fields, or even add, remove, or change a parent, yet the child protocol and constructor need not change as long as the parent protocol does not change.

Additional details and options for `define-record-type` are given in its formal description below.

**syntax:** `(define-record-type record-name clause ...)`

**syntax:** `(define-record-type (record-name constructor pred) clause ...)`

**libraries:** `(rnrs records syntactic),(rnrs)`

A `define-record-type` form, or record definition, is a definition and may appear anywhere other definitions may appear. It defines a record type identified by `record-name`, plus a predicate, constructor, accessors, and mutators for the record type. If the record definition takes the first form above, the names of the constructor and predicate are derived from `record-name:make-record-name` for the constructor and `record-name?` for the predicate. If the record definition takes the second form above, the name of the
constructor is constructor and the name of the predicate is pred. All names defined by a record
definition are scoped where the record definition appears.

The clauses clause ... of the record definition determine the fields of the record type and the names of
their accessors and mutators; its parent type, if any; its construction protocol; whether it is nongenerative and,
if so, whether its uid is specified; whether it is sealed; and whether it is opaque. The syntax and impact of each
clause is described below.

None of the clauses is required; thus, the simplest record definition is

(define-record-type record-name)

which defines a new, generative, non-sealed, non-opaque record type with no parent and no fields, plus a
constructor of no arguments and a predicate.

At most one of each kind of clause may be present in the set of clauses, and if a parent clause is present, a
parent-rtd clause must not be present. The clauses that appear may appear in any order.

Fields clause. A (fields field-spec ...) clause declares the fields of the record type. Each
field-spec must take one of the following forms:

field-name
(immmutable field-name)
(mutable field-name)
(immmutable field-name accessor-name)
(mutable field-name accessor-name mutator-name)

where field-name, accessor-name, and mutator-name are identifiers. The first form,
field-name, is equivalent to (immmutable field-name). The value of a field declared immmutable
may not be changed, and no mutator is created for it. With the first three forms, the name of the accessor is
rname-fname, where rname is the record name and fname is the field name. With the third form, the
name of the accessor is rname-fname-set!. The fourth and fifth forms explicitly declare the accessor and
mutator names.

If no fields clause is present or the list field-spec ... is empty, the record type has no fields (other
than parent fields, if any).

Parent clause. A (parent parent-name) clause declares the parent record type; parent-name must
be the name of a non-sealed record type previously defined via define-record-type. Instances of a
record type are also considered instances of its parent record type and have all the fields of its parent record
type in addition to those declared via the fields clause.

Nongenerative clause. A nongenerative clause may take one of two forms:

(nongenerative)
(nongenerative uid)

where uid is a symbol. The first form is equivalent to the second, with a uid generated by the implementation
at macro-expansion time. When a define-record-type form with a nongenerative clause is evaluated, a
new type is created if and only if the uid is not the uid of an existing record type.

If it is the uid of an existing record type, the parent, field-names, sealed property, and opaque property must
match as follows.
If a parent is specified, the existing record type must have the same parent rtd (by eqv?). If a parent is not specified, the existing record type must not have a parent.

The same number of fields must be provided, with the same names and in the same order, and the mutability of each field must be the same.

If a (sealed #t) clause is present, the existing record type must be sealed. Otherwise, the existing record type must not be sealed.

If an (opaque #t) clause is present, the existing record type must be opaque. Otherwise, the existing record type must be opaque if and only if an opaque parent type is specified.

If these constraints are met, no new record type is created, and the other products of the record-type definition (constructor, predicate, accessors, and mutators) operate on records of the existing type. If these constraints are not met, the implementation may treat it as a syntax violation, or it may raise a run-time exception with condition type &assertion.

With the first form of nongenerative clause, the generated uid can be the uid of an existing record type only if the same definition is executed multiple times, e.g., if it appears in the body of a procedure that is invoked multiple times.

If uid is not the uid of an existing record type, or if no nongenerative clause is present, a new record type is created.

Protocol clause. A (protocol expression) determines the protocol that the generated constructor uses to construct instances of the record type. It must evaluate to a procedure, and this procedure should be an appropriate protocol for the record type, as described on page 326.

Sealed clause. A sealed clause of the form (sealed #t) declares that the record type is sealed. This means that it cannot be extended, i.e., cannot be used as the parent for another record definition or make-record-type-descriptor call. If no sealed clause is present or if one of the form (sealed #f) is present, the record type is not sealed.

Opaque clause. An opaque clause of the form (opaque #t) declares that the record type is opaque. Instances of an opaque record type are not considered records by the record? predicate or, more importantly, the rtd-extraction procedure record-rtd, which are both described in Section 9.3. Thus, it is not possible for code that does not have access to the record-name, accessors, or mutators to access or modify any of the fields of an opaque record type. A record type is also opaque if its parent is opaque. If no opaque clause is present or if one of the form (opaque #f) is present, and the parent, if any, is not opaque, the record type is not opaque.

Parent-rtd clause. A (parent-rtd parent-rtd parent-rcd) clause is an alternative to the parent clause for specifying the parent record type, along with a parent record constructor descriptor. It is primarily useful when the parent rtd and rcd were obtained via calls to make-record-type-descriptor and make-record-constructor-descriptor.

parent-rtd must evaluate to an rtd or #f. If parent-rtd evaluates to #f, parent-rcd must also evaluate to #f. Otherwise, parent-rcd must evaluate to an rcd or #f. If parent-rcd evaluates to an rcd, it must encapsulate an rtd equivalent (by eqv?) to the value of parent-rtd. If the value of parent-rcd is #f, it is treated as an rcd for the value of parent-rtd with a default protocol.

The define-record-type form is designed in such a way that it is normally possible for a compiler to determine the shapes of the record types it defines, including the offsets for all fields. This guarantee does not hold, however, when the parent-rtd clause is used, since the parent rtd might not be determinable until run time. Thus, the parent clause is preferred over the parent-rtd clause whenever the parent clause suffices.
These identifiers are auxiliary keywords for `define-record-type`. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

### Section 9.2. Procedural Interface

The procedural `(make-record-type-descriptor)` interface may also be used to create new record types. The procedural interface is more flexible than the syntactic interface, but this flexibility can lead to less readable and efficient programs, so programmers should use the syntactic interface whenever it suffices.

**procedure:** `(make-record-type-descriptor name parent uid s? o? fields)`

**returns:** a record-type descriptor (rtd) for a new or existing record type

**libraries:** `(rnrs records procedural), (rnrs)`

*name* must be a symbol, *parent* must be `#f` or the rtd of a non-sealed record type, *uid* must be `#f` or a symbol, and *fields* must be a vector, each element of which is a two-element list of the form `(mutable field-name)` or `(immutable field-name)`. The field names `field-name ...` must be symbols and need not be distinct from each other.

If *uid* is `#f` or is not the uid of an existing record type, this procedure creates a new record type and returns a record-type descriptor (rdt) for the new type. The type has the parent type (page 325) described by *parent*, if nonfalse; the uid specified by *uid*, if nonfalse; and the fields specified by *fields*. It is sealed (page 330) if *s?* is nonfalse. It is opaque (page 330) if *opaque* is nonfalse or the parent (if specified) is opaque. The name of the new record type is *name* and the names of the fields are `field-name ...`.

If *uid* is nonfalse and is the uid (page 325) of an existing record type, the *parent*, *fields*, *s?*, and *o?* arguments must match the corresponding characteristics of the existing record type. That is, *parent* must be the same (by `eqv?`); the number of fields must be the same; the fields must have the same names, be in the same order, and have the same mutability; *s?* must be false if and only if the existing record type is sealed; and, if a parent is not specified or is not opaque, *o?* must be false if and only if the existing record type is opaque. If this is the case, `make-record-type-descriptor` returns the rtd for the existing record type. Otherwise, an exception with condition type `&assertion` is raised.

Using the rtd returned by `make-record-type-descriptor`, programs can generate constructors, type predicates, field accessors, and field mutators dynamically. The following code demonstrates how the procedural interface might be used to create a `point` record type and associated definitions similar to those of the second `point` record definition in Section 9.1, with a mutable `x` field and an immutable `y` field.

```
(define point-rtd (make-record-type-descriptor 'point #f #f #f '#((mutable x) (immutable y))))
(define point-rcd (make-record-constructor-descriptor point-rtd #f #f))
(define make-point (record-constructor point-rcd))
(define point? (record-predicate point-rtd))
```
(define point-x (record-accessor point-rtd 0))
(define point-y (record-accessor point-rtd 1))
(define point-x-set! (record-mutator point-rtd 0))

See the additional examples given at the end of this section.

procedure: (record-type-descriptor? obj)
returns: #f if obj is a record-type descriptor, otherwise #f
libraries: (rnrs records procedural), (rnrs)

See the examples given at the end of this section.

procedure: (make-record-constructor-descriptor rtd parent-rcd protocol)
returns: a record-constructor descriptor (rcd)
libraries: (rnrs records procedural), (rnrs)

An rtd alone is sufficient to create predicates, accessors, and mutators. To create a constructor, however, it is first necessary to create a record-constructor descriptor (rcd) for the record type. An rcd encapsulates three pieces of information: the rtd of the record type for which the rcd has been created, the parent rcd (if any), and the protocol.

The parent-rcd argument must be an rcd or #f. If it is an rcd, rtd must have a parent rtd, and the parent rtd must be the same as the rtd encapsulated within parent-rcd. If parent-rcd is false, either rtd has no parent or an rcd with a default protocol is assumed for the parent.

The protocol argument must be a procedure or #f. If it is #f, a default protocol is assumed. Protocols are discussed on page 326.

See the examples given at the end of this section.

syntax: (record-type-descriptor record-name)
returns: the rtd for the record type identified by record-name
syntax: (record-constructor-descriptor record-name)
returns: the rcd for the record type identified by record-name
libraries: (rnrs records syntactic), (rnrs)

Each record definition creates, behind the scenes, an rtd and rcd for the defined record type. These procedures allow the rtd and rcd to be obtained and used like any other rtd or rcd. record-name must be the name of a record previously defined via define-record-type.

procedure: (record-constructor rcd)
returns: a record constructor for the record type encapsulated within rcd
libraries: (rnrs records procedural), (rnrs)

The behavior of the record constructor is determined by the protocol and parent rcd (if any) also encapsulated within rcd.

See the examples given at the end of this section.

procedure: (record-predicate rtd)
returns: a predicate for rtd
libraries: (rnrs records procedural), (rnrs)

This procedure returns a predicate that accepts one argument and returns #t if the argument is an instance of
the record-type described by \( rtd, #f \) otherwise.

See the examples given at the end of this section.

**procedure**: (record-accessor \( rtd \) \( idx \))  
**returns**: an accessor for the field of \( rtd \) specified by \( idx \)  
**libraries**: (rnrs records procedural), (rnrs)

\( idx \) must be a nonnegative integer less than the number of fields of \( rtd \), not counting parent fields. An \( idx \) value of 0 specifies the first field given in the define-record-type form or make-record-type-descriptor call that created the record type, 1 specifies the second, and so on.

A child \( rtd \) cannot be used directly to create accessors for parent fields. To create an accessor for a parent field, the record-type descriptor of the parent must be used instead.

See the examples given at the end of this section.

**procedure**: (record-mutator \( rtd \) \( idx \))  
**returns**: a mutator for the field of \( rtd \) specified by \( idx \)  
**libraries**: (rnrs records procedural), (rnrs)

\( idx \) must be a nonnegative integer less than the number of fields of \( rtd \), not counting parent fields. An \( idx \) value of 0 specifies the first field given in the define-record-type form or make-record-type-descriptor call that created the record type, 1 specifies the second, and so on.

The indicated field must be mutable; otherwise, an exception with condition type \&assertion is raised.

A child \( rtd \) cannot be used directly to create mutators for parent fields. To create a mutator for a parent field, the record-type descriptor of the parent must be used instead.

The following example illustrates the creation of parent and child record types, predicates, accessors, mutators, and constructors using the procedures described in this section.

```
(define rtd/parent
  (make-record-type-descriptor 'parent #f #f #f #f
    '##((mutable x))))

(record-type-descriptor? rtd/parent) ⇒ #t
(define parent? (record-predicate rtd/parent))
(define parent-x (record-accessor rtd/parent 0))
(define set-parent-x! (record-mutator rtd/parent 0))

(define rtd/child
  (make-record-type-descriptor 'child rtd/parent #f #f #f
    '##((mutable x) (immutable y))))

(define child? (record-predicate rtd/child))
(define child-x (record-accessor rtd/child 0))
(define set-child-x! (record-mutator rtd/child 0))
(define child-y (record-accessor rtd/child 1))

(record-mutator rtd/child 1) ⇒ exception: immutable field

(define rcd/parent
  (make-record-constructor-descriptor rtd/parent #f
```
Section 9.3. Inspection

This section describes various procedures for asking questions about or extracting information from record-type descriptors (rtds). It also describes the record-rtd procedure, with which the rtd of a non-opaque record instance may be extracted, allowing the record type of the instance to be inspected and, via record accessors and mutators generated from the rtd, the record itself to be inspected or modified. This is a powerful feature that permits the coding of portable record printers and inspectors.

The record-type descriptor cannot be extracted from an instance of an opaque record type; this is the feature that distinguishes opaque from non-opaque record types.

procedure: (record-type-name rtd)  
returns: the name associated with rtd  
libraries: (rnrs records inspection), (rnrs)

(define record->name  
  (lambda (x)  
    (and (record? x) (record-type-name (record-rtd x))))))

(define-record-type dim (fields w l h))  
(record->name (make-dim 10 15 6)) ⇒ dim

(define-record-type dim (fields w l h) (opaque #t))  
(record->name (make-dim 10 15 6)) ⇒ #f
procedure: (record-type-parent rtd)
returns: the parent of rtd, or #f if it has no parent
libraries: (rnrs records inspection), (rnrs)

(define-record-type point (fields x y))
(define-record-type cpoint (parent point) (fields color))
(record-type-parent (record-type-descriptor point)) ⇒ #f
(record-type-parent (record-type-descriptor cpoint)) ⇒ #<rtd>

procedure: (record-type-uid rtd)
returns: the uid of rtd, or #f if it has no uid
libraries: (rnrs records inspection), (rnrs)

Whether a record type created without a programmer-supplied uid actually has one anyway is left up to the implementation, so this procedure is never guaranteed to return #f.

(define-record-type point (fields x y))
(define-record-type cpoint
    (parent point)
    (fields color)
    (nongenerative e40cc926-8cf4-4559-a47c-cac636630314))
(record-type-uid (record-type-descriptor point)) ⇒ unspecified
(record-type-uid (record-type-descriptor cpoint)) ⇒ e40cc926-8cf4-4559-a47c-cac636630314

procedure: (record-type-generative? rtd)
returns: #t if the record type described by rtd is generative, #f otherwise
procedure: (record-type-sealed? rtd)
returns: #t if the record type described by rtd is sealed, #f otherwise
procedure: (record-type-opaque? rtd)
returns: #t if the record type described by rtd is opaque, #f otherwise
libraries: (rnrs records inspection), (rnrs)

(define-record-type table
    (fields keys vals)
    (opaque #t))
(define rtd (record-type-descriptor table))
(record-type-generative? rtd) ⇒ #t
(record-type-sealed? rtd) ⇒ #f
(record-type-opaque? rtd) ⇒ #t

(define-record-type cache-table
    (parent table)
    (fields key val)
    (nongenerative))
(define rtd (record-type-descriptor cache-table))
(record-type-generative? rtd) ⇒ #f
(record-type-sealed? rtd) ⇒ #f
(record-type-opaque? rtd) ⇒ #t

procedure: (record-type-field-names rtd)
returns: a vector containing the names of the fields of the type described by rtd
libraries: (rnrs records inspection), (rnrs)
The vector returned by this procedure is immutable: the effect on \( rtd \) of modifying it is unspecified. The vector does not include parent field names. The order of the names in the vector is the same as the order in which the fields were specified in the \( \text{define-record-type} \) form or \( \text{make-record-type-descriptor} \) call that created the record type.

\[
\begin{align*}
&(\text{define-record-type} \ \text{point} \ (\text{fields} \ x \ y)) \\
&(\text{define-record-type} \ \text{cpoint} \ (\text{parent} \ \text{point}) \ (\text{fields} \ \text{color})) \\
&(\text{record-type-field-names} \\
&\quad (\text{record-type-descriptor} \ \text{point})) \Rightarrow \#(x \ y) \\
&(\text{record-type-field-names} \\
&\quad (\text{record-type-descriptor} \ \text{cpoint})) \Rightarrow \#(\text{color})
\end{align*}
\]

**procedure**: \((\text{record-field-mutable?} \ rtd \ idx)\)

**returns**: \#t if the specified field of \( rtd \) is mutable, \#f otherwise

**libraries**: (rnrs records inspection), (rnrs)

\( idx \) must be a nonnegative integer less than the number of fields of \( rtd \), not counting parent fields. An \( idx \) value of 0 specifies the first field given in the \( \text{define-record-type} \) form or \( \text{make-record-type-descriptor} \) call that created the record type, 1 specifies the second, and so on.

\[
\begin{align*}
&(\text{define-record-type} \ \text{point} \ (\text{fields} \ (\text{mutable} \ x) \ (\text{mutable} \ y))) \\
&(\text{define-record-type} \ \text{cpoint} \ (\text{parent} \ \text{point}) \ (\text{fields} \ \text{color}))
\end{align*}
\]

\[(\text{record-field-mutable?} \ (\text{record-type-descriptor} \ \text{point}) \ 0) \Rightarrow \#t \quad (\text{record-field-mutable?} \ (\text{record-type-descriptor} \ \text{cpoint}) \ 0) \Rightarrow \#f\]

**procedure**: \((\text{record?} \ \text{obj})\)

**returns**: \#t if \( \text{obj} \) is a non-opaque record instance, \#f otherwise

**libraries**: (rnrs records inspection), (rnrs)

When passed an instance of an opaque record type, \( \text{record?} \) returns \#f. While an instance of an opaque record type is, in essence, a record, the point of opacity is to hide all representation information from the parts of a program that should not have access to the information, and this includes whether an object is a record. Furthermore, the primary purpose of this predicate is to allow programs to check whether it is possible to obtain from the argument an \( rtd \) via the \( \text{record-rtd} \) procedure described below.

\[
\begin{align*}
&(\text{define-record-type} \ \text{statement} \ (\text{fields} \ \text{str})) \\
&(\text{define} \ q \ (\text{make-statement} \ "\text{He's dead, Jim}")) \\
&(\text{statement?} \ q) \Rightarrow \#t \\
&(\text{record?} \ q) \Rightarrow \#t
\end{align*}
\]

\[
\begin{align*}
&(\text{define-record-type} \ \text{opaque-statement} \ (\text{fields} \ \text{str}) \ (\text{opaque} \ \#t)) \\
&(\text{define} \ q \ (\text{make-opaque-statement} \ "\text{He's moved on, Jim}")) \\
&(\text{opaque-statement?} \ q) \Rightarrow \#t \\
&(\text{record?} \ q) \Rightarrow \#f
\end{align*}
\]

**procedure**: \((\text{record-rtd} \ \text{record})\)

**returns**: the record-type descriptor (\( rtd \)) of \( \text{record} \)

**libraries**: (rnrs records inspection), (rnrs)

The argument must be an instance of a non-opaque record type. In combination with some of the other procedures described in this section and Section 9.2, \( \text{record-rtd} \) allows the inspection or mutation of record instances, even if the type of the instance is unknown to the inspector. This capability is illustrated by the procedure \( \text{print-fields} \) below, which accepts a record argument and writes the name and value of
each field of the record.

```scheme
(define print-fields
  (lambda (r)
    (unless (record? r)
      (assertion-violation 'print-fields "not a record" r))
    (let loop ([rtd (record-rtd r)])
      (let ([prtd (record-type-parent rtd)])
        (when prtd (loop prtd))
      (let* ([v (record-type-field-names rtd)]
            [n (vector-length v)])
        (do ([i 0 (+ i 1)])
          (when (= i n)
            (write (vector-ref v i))
            (display " =")
            (write ((record-accessor rtd i) r))
            (newline)))))))
```

With the familiar definitions of `point` and `cpoint`:

```scheme
(define-record-type point (fields x y))
(define-record-type cpoint (parent point) (fields color))
```

the expression `(print-fields (make-cpoint -3 7 'blue))` displays the following three lines.

```
x=-3
y=7
color=blue
```
Chapter 10. Libraries and Top-Level Programs

Libraries and top-level programs are the basic units of portable code in the language defined by the Revised\textsuperscript{6} Report on Scheme [24]. Top-level programs may import from one or more libraries, and libraries may import from other libraries.

Libraries are named using a parenthesized syntax that encloses a sequence of identifiers, optionally followed by a version; the version is itself a parenthesized form that encloses a sequence of subversions represented as exact nonnegative integers. So, for example, (a), (a b), (a b ()), and (a b (1 2 3)) are all valid library names. Implementations typically treat the sequence of names as a path by which a library's source or object code can be found, possibly rooted in some standard set of locations in the host-machine's filesystem.

An implementation of the standard library mechanism is available with the portable implementation of syntax-case at http://www.cs.indiana.edu/syntax-case/.

Section 10.1. Standard Libraries

The Revised\textsuperscript{6} Report [24] describes a base library

\[(rnrs base (6))\]

that defines the most commonly used features of the language. A separate Standard Libraries document [26] describes the libraries listed below.

\[(rnrs arithmetic bitwise (6))\]
\[(rnrs arithmetic fixnums (6))\]
\[(rnrs arithmetic flonums (6))\]
\[(rnrs bytevectors (6))\]
\[(rnrs conditions (6))\]
\[(rnrs control (6))\]
\[(rnrs enums (6))\]
\[(rnrs eval (6))\]
\[(rnrs exceptions (6))\]
\[(rnrs files (6))\]
\[(rnrs hashtables (6))\]
\[(rnrs io ports (6))\]
\[(rnrs io simple (6))\]
\[(rnrs lists (6))\]
\[(rnrs mutable-pairs (6))\]
\[(rnrs mutable-strings (6))\]
\[(rnrs programs (6))\]
\[(rnrs r5rs (6))\]
\[(rnrs records procedural (6))\]
\[(rnrs records syntactic (6))\]
\[(rnrs records inspection (6))\]
\[(rnrs sorting (6))\]
\[(rnrs syntax-case (6))\]
\[(rnrs unicode (6))\]

One more library is described in the Standard Libraries document, a composite library

\[(rnrs (6))\]
that exports all of the (rnrs base (6)) bindings along with those of the other libraries listed above, except those of (rnrs eval (6)), (rnrs mutable-pairs (6)), (rnrs mutable-strings (6)), and (rnrs r5rs (6)).

Although each of these libraries has the version (6), references to them can and in most cases should leave the version out, e.g., the composite library should be referenced simply as (rnrs).

Section 10.2. Defining New Libraries

New libraries are defined with the library form, which has the following syntax.

(library library-name
  (export export-spec ...)
  (import import-spec ...)
  library-body)

A library-name specifies the name and possibly version by which the library is identified by the import form of another library or top-level program. It also serves as kind of path that the implementation uses to locate the library, via some implementation-specific process, whenever it needs to be loaded. A library-name has one of the following two forms:

(identifier identifier ...)
(identifier identifier ... version)

where version has the following form:

(subversion ...)

and each subversion represents an exact nonnegative integer. A library name with no version is treated the same as a library name with the empty version (). For example, (list-tools setops) and (list-tools setops ()) are equivalent and specify a library name with no version, while (list-tools setops (1 2)) specifies a versioned library name, which can be thought of as Version 1.2 of the (list-tools setops) library.

The export subform names the exports and, optionally, the names by which they should be known outside of the library. Each export-spec takes one of the following two forms:

identifier
(rename (internal-name export-name) ...)

where each internal-name and export-name is an identifier. The first form names a single export, identifier, whose export name is the same as its internal name. The second names a set of exports, each of whose export name is given explicitly and may differ from its internal name.

The import subform names the other libraries upon which the new library depends and, possibly, the set of identifiers to be imported and the names by which they should be known inside the new library. It may also specify when the bindings should be made available for implementations that require such information. Each import-spec takes one of the following two forms:

import-set
(for import-set import-level ...)

where import-level is one of the following:
run
expand
(meta level)

and level represents an exact integer.

The for syntax declares when the imported bindings might be used by the importing library and thus when the implementation must make the bindings available. run and (meta 0) are equivalent and specify that the bindings imported from a library might be referenced by the run-time expressions (define right-hand-side expressions and initialization expressions) of the importing library. expand and (meta 1) are equivalent and specify that the bindings imported from a library might be referenced by the transformer expressions (define-syntax, let-syntax, or letrec-syntax right-hand-side expressions) of the importing library. (meta 2) specifies that the bindings imported from a library might be referenced by a transformer expression that appears within a transformer expression of the importing library, and so on for higher meta levels. Negative meta levels may also be specified and are needed in certain circumstances when a transformer expands into the transformer for another keyword binding used at a lower meta level.

A library export may have a non-zero export meta level, in which case the effective import level is the sum of the level specified by for and the export level. The exports of each standard library except (rnrs base) and (rnrs) have export level zero. For (rnrs base), all exports have export level zero except for syntax-rules, identifier-syntax, and their auxiliary keywords _, ..., and set!. set! has export levels zero and one, while the others have export level one. All exports of the (rnrs) library have export levels zero and one.

It can be difficult for the programmer to specify the import levels that allow a library or top-level program to compile or run properly. Moreover, it is often impossible to cause a library's bindings to be made available when they are needed without causing them to be made available in some cases when they are not needed. For example, it is not possible to say that the run-time bindings of a library A are needed when a library B is expanded without also having the run-time bindings of A made available when code importing B is expanded. Making bindings available involves executing the code for the right-hand sides of the bindings and possibly executing initialization expressions as well, so the inability to specify when bindings are needed precisely can add both compile- and run-time overhead to a program.

Because of this, implementations are permitted to ignore export levels and the for wrapper on an import-set and instead automatically determine, while expanding an importing library or top-level program, when an imported library's bindings must be made available, based on where references to the imported library's exports actually appear. When using such an implementation, the for wrapper need never be used, i.e., all import-specs can be import-sets. If code is intended for use with systems that do not automatically determine when a library's bindings must be made available, however, the for must be used if the importing library's bindings would not otherwise be available at the right time.

An import-set takes one of the following forms:

library-spec
(only import-set identifier ...)
(except import-set identifier ...)
(prefix import-set prefix)
(rename import-set (import-name internal-name) ...)

where prefix, import-name, and internal-name are identifiers. An import-set is a recursive specification of the identifiers to be imported from a library and possibly the names by which they should be known within the importing library. At the base of the recursive structure must sit a library-spec, which identifies a library and imports all of the identifiers from that library. An only wrapper restricts the imported identifiers of the enclosed import-set to the ones listed, an except wrapper restricts the imported
identifiers of the enclosed import-set to those not listed, a prefix wrapper adds a prefix to each of the imported identifiers of the enclosed import-set, and a rename wrapper specifies internal names for selected identifiers of the enclosed import-set, while leaving the names of the other imports alone. So, for example, the import-set

(prefix
  (only
    (rename (list-tools setops) (difference diff))
    union
    diff)
  set:)

imports only union and difference from the (list-tools setops) library, renames difference to diff while leaving union alone, and adds the prefix set: to the two names so that the names by which the two imports are known inside the importing library are set:union and set:diff.

A library-spec takes one of the following forms:

library-reference
(library library-reference)

where a library-reference is in either of the following two forms:

(identifier identifier ...)
(identifier identifier ... version-reference)

Enclosing a library-reference in a library wrapper is necessary when the first identifier of the library-reference is for, library, only, except, prefix, or rename, to distinguish it from an import-spec or import-set identified by one of these keywords.

A version-reference identifies a particular version of the library or a set of possible versions. A version-reference has one of the following forms:

(subversion-reference₁ ... subversion-referenceₙ)
(and version-reference ...)
(or version-reference ...)
(not version-reference)


A subversion-reference takes one of the following forms:

subversion
(>= subversion)
(<= subversion)
(and subversion-reference ...)
(or subversion-reference ...)
(not subversion-reference)
A subversion-reference of the first form matches a subversion if it is identical to it. A $\geq$ subversion-reference matches a version's subversion if the version's subversion is greater than or equal to the subversion appearing within the $\geq$ form. Similarly, a $\leq$ subversion-reference matches a version's subversion if the version's subversion is less than or equal to the subversion appearing within the $\geq$ form. An and subversion-reference form matches a version's subversion if each of its subversion-reference subforms matches the version's subversion. An or subversion-reference matches a version's subversion if any of its subversion-reference subforms match the version's subversion. A not subversion-reference matches a version's subversion if its subversion-reference subform does not match the version's subversion.

For example, if two versions of a library are available, one with version (1 2) and the other with version (1 3 1), the version references () and (1) match both, (1 2) matches the first but not the second, (1 3) matches the second but not the first, (1 ($\geq$ 2)) matches both, and (and (1 ($\geq$ 3)) (not (1 3 1))) matches neither.

When a library reference identifies more than one available library, one of the available libraries is selected in some implementation-dependent manner.

Libraries and top-level programs should not, directly or indirectly, specify the import of two libraries that have the same names but different versions. To avoid problems such as incompatible types and replicated state, implementations are encouraged, though not required, to prohibit programs from importing two versions of the same library.

A library-body contains definitions of exported identifiers, definitions of identifiers not intended for export, and initialization expressions. It consists of a (possibly empty) sequence of definitions followed by a (possibly empty) sequence of initialization expressions. When begin, let-syntax, or letrec-syntax forms occur in a library body prior to the first expression, they are spliced into the body. Any body form may be produced by a syntactic extension, including definitions, the splicing forms just mentioned, or initialization expressions. A library body is expanded in the same manner as a lambda or other body (page 292), and it expands into the equivalent of a letrec* form so that the definitions and initialization forms in the body are evaluated from left to right.

Each of the exports listed in a library's export form must either be imported from another library or defined within the library-body, in either case with the internal rather than the export name, if the two differ.

Each identifier imported into or defined within a library must have exactly one binding. If imported into a library, it must not be defined in the library body, and if defined in the library body, it must be defined only once. If imported from two libraries, it must have the same binding in both cases, which can happen only if the binding originates in one of the two libraries and is reexported by the other or if the binding originates in a third library and is reexported by both.

The identifiers defined within a library and not exported by the library are not visible in code that appears outside of the library. A syntactic extension defined within a library may, however, expand into a reference to such an identifier, so that the expanded code does contain a reference to the identifier; this is referred to as an indirect export.

The exported variables of a library are immutable both inside the library and outside, whether they are explicitly or implicitly exported. It is a syntax violation if an explicitly exported variable appears on the left-hand side of a set! expression within or outside of the exporting library. It is also a syntax violation if any other variable defined by a library appears on the left-hand side of a set! expression and is indirectly exported.

Libraries are loaded and the code contained within them evaluated on an "as needed" basis by the
implementation, as determined by the import relationships among libraries. A library's transformer expressions (the expressions on the right-hand sides of a library body's define-sytax forms) may be evaluated at different times from the library's body expressions (the expressions on the right-hand side of the body's define forms, plus initialization expressions). At a minimum, the transformer expressions of a library must be evaluated when (if not before) a reference to one of the library's exported keywords is found while expanding another library or top-level program, and the body expressions must be evaluated when (if not before) a reference to one of the library's exported variables is evaluated, which may occur either when a program using the library is run or when another library or top-level program is being expanded, if the reference is evaluated by a transformer called during the expansion process. An implementation may evaluate a library's transformer and body expressions as many times as it pleases in the process of expanding other libraries. In particular, it may evaluate the expressions zero times if they are not actually needed, exactly one time, or one time for each meta level of the expansion. It is generally a bad idea for the evaluation of a library's transformer or body expressions to involve externally visible side effects, e.g., popping up a window, since the time or times at which these side effects occur is unspecified. Localized effects that affect only the library's initialization, e.g., to create a table used by the library, are generally okay.

Examples are given in Section 10.4.

Section 10.3. Top-Level Programs

A top-level program is not a syntactic form per se but rather a set of forms that are usually delimited only by file boundaries. Top-level programs can be thought of as library forms without the library wrapper, library name, and export form. The other difference is that definitions and expressions can be intermixed within the body of a top-level program but not within the body of a library. Thus the syntax of a top-level program is, simply, an import form followed by a sequence of definitions and expressions:

```
(import import-spec ...)  
definition-or-expression  
...                      
```

An expression that appears within a top-level program body before one or more definitions is treated as if it appeared on the right-hand side of a definition for a dummy variable that is not visible anywhere within the program.

**procedure:** (command-line)  
**returns:** a list of strings representing command-line arguments  
**libraries:** (rnrs programs),(rnrs)

This procedure may be used within a top-level program to obtain a list of the command-line arguments passed to the program.

**procedure:** (exit)  
**procedure:** (exit obj)  
**returns:** does not return  
**libraries:** (rnrs programs),(rnrs)

This procedure may be used to exit from a top-level program to the operating system. If no obj is given, the exit value returned to the operating system should indicate a normal exit. If obj is false, the exit value returned to the operating system should indicate an abnormal exit. Otherwise, obj is translated into an exit value as appropriate for the operating system.
Section 10.4. Examples

The example below demonstrates several features of the library syntax. It defines "Version 1" of the \texttt{(list-tools setops)} library, which exports two keywords and several variables. The library imports the \texttt{(rnrs base)} library, which provides everything it needs except the \texttt{member} procedure, which it imports from \texttt{(rnrs lists)}. Most of the variables exported by the library are bound to procedures, which is typical.

The syntactic extension \texttt{set} expands into a reference to the variable \texttt{list->set}, and \texttt{member?} similarly expands into a reference to the variable \texttt{$member?}. While \texttt{list->set} is explicitly exported, \texttt{$member?} is not. This makes \texttt{$member?} an indirect export. The procedure \texttt{u-d-help} is not explicitly exported, and since neither of the exported syntactic extensions expand into references to \texttt{u-d-help}, it is not indirectly exported either. This means it could be assigned, but it is not assigned in this example.

```scheme
(library (list-tools setops 1))
  (export set empty-set empty-set? list->set set->list
    union intersection difference member?)
  (import (rnrs base) (only (rnrs lists) member))

  (define-syntax set
    (syntax-rules ()
      [(_ x ...) (list->set (list x ...))]))

  (define empty-set '())

  (define empty-set? null?)

  (define list->set
    (lambda (ls)
      (cond
        [(null? ls) '()]
        [(member? (car ls) (cdr ls)) (list->set (cdr ls))]
        [else (cons (car ls) (list->set (cdr ls)))])))

  (define set->list (lambda (set) set))

  (define u-d-help
    (lambda (s1 s2 ans)
      (let f ([s1 s1])
        (cond
          [(null? s1) ans]
          [(member? (car s1) s2) (f (cdr s1))]
          [else (cons (car s1) (f (cdr s1)))])))

  (define union
    (lambda (s1 s2)
      (u-d-help s1 s2 s2)))

  (define intersection
    (lambda (s1 s2)
      (cond
        [(null? s1) '()]
        [(member? (car s1) s2) (f (cdr s1))]
        [else (cons (car s1) (f (cdr s1)))]))
```

The Scheme Programming Language, 4th Edition

Section 10.4. Examples
The next library, (more-setops), defines a few additional set operations in terms of the (list-tools setops) operations. No version is included in the library reference to (list-tools setops); this is equivalent to an empty version reference, which matches any version. The quoted-set keyword is interesting because its transformer references list->set from (list-tools setops) at expansion time. As a result, if another library or top-level program that imports from (more-setops) references quoted-set, the run-time expressions of the (list-tools setops) library will have to be evaluated when the other library or top-level program is expanded. On the other hand, the run-time expressions of the (list-tools setops) library need not be evaluated when the (more-setops) library is itself expanded.

(library (more-setops)
  (export quoted-set set-cons set-remove)
  (import (list-tools setops) (rnrs base) (rnrs syntax-case))

(define-syntax quoted-set
  (lambda (x)
    (syntax-case x ()
      [(k elt ...) #`(quote #,(datum->syntax #'k (list->set
        (syntax->datum #'(elt ...))))))]))

(define set-cons
  (lambda (opt optset)
    (union (set opt) optset)))

(define set-remove
  (lambda (opt optset)
    (difference optset (set opt)))))

If the implementation does not automatically infer when bindings need to be made available, the import form in the (more-setops) library must be modified to specify at which meta levels the bindings it imports are used via the for import-spec syntax as follows:

(import
  (for (list-tools setops) expand run)
To complete the example, the short top-level program below exercises several of the
(list-tools setops) and (more-setops) exports.

```
(import (list-tools setops) (more-setops) (rnrs))
(define-syntax pr
  (syntax-rules ()
    [(_ obj)
      (begin
        (write 'obj)
        (display " ;=> ")
        (write obj)
        (newline))])))
(define get-set1
  (lambda ()
    (quoted-set a b c d)))
(define set1 (get-set1))
(define set2 (quoted-set a c e))

(pr (list set1 set2))
(pr (eq? (get-set1) (get-set1)))
(pr (eq? (get-set1) (set 'a 'b 'c 'd)))
(pr (union set1 set2))
(pr (intersection set1 set2))
(pr (difference set1 set2))
(pr (set-cons 'a set2))
(pr (set-cons 'b set2))
(pr (set-remove 'a set2))
```

What running this program should print is left as an exercise for the reader.

Additional library and top-level program examples are given in Chapter 12.
Chapter 11. Exceptions and Conditions

*Exceptions* and *conditions* provide the means for system and user code to signal, detect, and recover from errors that occur when a program is run.

Exceptions are raised by the standard syntactic forms and procedures under a variety of circumstances, e.g., when the wrong number of arguments is passed to a procedure, when the syntax of an expression passed to `eval` is incorrect, or when a file cannot be opened by one of the file open procedures. In these situations, the exception is raised with a standard condition type.

Exceptions may also be raised by user code via the `raise` or `raise-continuable` procedures. In this case, the exception may be raised with one of the standard condition types, a user-defined subtype of one of the standard condition types (possibly defined using `define-condition-type`), or an arbitrary Scheme value that is not a condition type.

At any point during a program's execution, a single exception handler, called the *current exception handler*, is charged with handling all exceptions that are raised. By default, the current exception handler is one provided by the implementation. The default exception handler typically prints a message that describes the condition or other value with which the exception was raised and, for any serious condition, terminates the running program. In interactive systems, this typically means a reset to the read-eval-print loop.

User code may establish a new current exception handler via the `guard` syntax or the `with-exception-handler` procedure. In either case, the user code may handle all exceptions or, based on the condition or other value with which the exception was raised, just some of the exceptions while reraising the others for the old current exception handler to handle. When `guard` forms and `with-exception-handler` calls are nested dynamically, a chain of exception handlers is established, and each may defer to the next in the chain.

**Section 11.1. Raising and Handling Exceptions**

**procedure**: (raise obj)

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

**returns**: see below

**libraries**: (rnrs exceptions), (rnrs)

Both of these procedures raise an exception, effectively invoking the current exception handler, passing *obj* as the only argument. For `raise`, the exception is *non-continuable*, while for `raise-continuable`, the exception is *continuable*. An exception handler may return (with zero or more values) to the continuation of a continuable exception. If an exception handler attempts to return to the continuation of a non-continuable exception, however, a new exception with condition type `&non-continuable` is raised. Thus, `raise` never returns, while `raise-continuable` may return zero or more values, depending upon the exception handler.

If the current exception handler, *p*, was established via a `guard` form or call to `with-exception-handler`, the current exception handler is reset to the handler that was current when *p* was established before `raise` or `raise-continuable` invokes *p*. This allows *p* to defer to the preexisting exception handler simply by reraising the exception, and it helps prevent infinite regression when an exception handler inadvertently causes a different exception to be raised. If *p* returns and the exception is continuable, *p* is reinstated as the current exception handler.

```scheme
(rise
  (condition
    (make-error)
    (make-message-condition "no go"))) ⇒ error: no go
```
(raise-continuable
 (condition
  (make-violation)
  (make-message-condition "oops"))) ⇒ violation: oops

(list
 (call/cc
  (lambda (k)
    (vector
     (with-exception-handler
      (lambda (x) (k (+ x 5)))
      (lambda () (+ (raise 17) 8)))))

(list
 (vector
  (with-exception-handler
   (lambda (x) (+ x 5))
   (lambda () (+ (raise-continuable 17) 8))))

(list
 (vector
  (with-exception-handler
   (lambda (x) (+ x 5))
   (lambda () (+ (raise 17) 8))))

violation: non-continuable

procedure: (error who msg irritant ...)
procedure: (assertion-violation who msg irritant ...)
libraries: (rnrs base), (rnrs)

error raises a non-continuable exception with condition type &error and should be used to describe situations for which the &error condition type is appropriate, typically a situation involving the program's interaction with something outside of the program. assertion-violation raises a non-continuable exception with condition type &assertion and should be used to describe situations for which the &assertion condition type is appropriate, typically an invalid argument to a procedure or invalid value of a subexpression of a syntactic form.

The continuation object with which the exception is raised also includes a &who condition whose who field is who if who is not #f, a &message condition whose message field is msg, and an &irritants condition whose irritants field is (irritant ...).

who must be a string, a symbol, or #f identifying the procedure or syntactic form reporting the error upon whose behalf the error is being reported. It is usually best to identify a procedure the programmer has called rather than some other procedure the programmer may not be aware is involved in carrying out the operation.

msg must be a string and should describe the exceptional situation. The irritants may be any Scheme objects and should include values that may have caused or been materially involved in the exceptional situation.

syntax: (assert expression)
returns: see below
libraries: (rnrs base), (rnrs)

assert evaluates expression and returns the value of expression if the value is not #f. If the value of expression is #f, assert raises a non-continuable exception with condition types &assertion and &message, with an implementation-dependent value in its message field. Implementations are encouraged to provide information about the location of the assert call within the condition whenever possible.

procedure: (syntax-violation who msg form)
procedure: (syntax-violation who msg form subform)
This procedure raises a non-continuable exception with a condition of type `&syntax`. It should be used to report a syntax error detected by the transformer of a syntactic extension. The value of the condition's form field is `form`, and the value of its subform field is `subform`, or `#f` if `subform` is not provided.

The continuation object with which the exception is raised also includes a `&who` condition whose who field is `who`, if `who` is not `#f` or is inferred from `form`, and a `&message` condition whose message field is `msg`.

`who` must be a string, a symbol, or `#f`. If `who` is `#f`, it is inferred to be the symbolic name of `form` if `form` is an identifier or the symbolic name of the first subform of `form` if `form` is a list-structured form whose first subform is an identifier. `message` must be a string. `form` should be the syntax object or datum representation of the syntactic form within which the syntax violation occurred, and `subform`, if not `#f`, should be a syntax object or datum representation of a subform more specifically involved in the violation. For example, if a duplicate formal parameter is found in a `lambda` expression, `form` might be the `lambda` expression and `subform` might be the duplicated parameter.

Some implementations attach source information to syntax objects, e.g., line, character, and filename for forms originating in a file, in which case this information might also be present as some implementation-dependent condition type within the condition object.

**procedure:** (with-exception-handler `procedure thunk`)  
**returns:** see below  
**libraries:** (rnrs exceptions), (rnrs)

This procedure establishes `procedure`, which should accept one argument, as the current exception handler in place of the old current exception handler, `old-proc`, and invokes `thunk` without arguments. If the call to `thunk` returns, `old-proc` is reestablished as the current exception handler and the values returned by `thunk` are returned. If control leaves or subsequently reenters the call to `thunk` via the invocation of a continuation obtained via `call/cc`, the procedure that was the current exception handler when the continuation was captured is reinstated.

```scheme
(define (try thunk)  
  (call/cc  
    (with-exception-handler  
      (lambda (x) (if (error? x) (k #f) (raise x))))  
    thunk)))
(try (lambda () 17)) ⇒ 17  
(try (lambda () (raise (make-error)))) ⇒ #f  
(try (lambda () (raise (make-violation)))) ⇒ violation  
(with-exception-handler  
  (lambda (x)  
    (raise  
      (make-message-condition "oops")  
      (simple-conditions x))))  
(lambda ()  
  (try (lambda () (raise (make-violation)))))) ⇒ violation: oops
```

**syntax:** (guard (var `clause_1` `clause_2` ...) `b1` `b2` ...)  
**returns:** see below  
**libraries:** (rnrs exceptions), (rnrs)
A guard expression establishes a new current exception handler, procedure (described below), in place of the old current exception handler, old-proc, and evaluates the body \( b_1 b_2 \ldots \). If the body returns, guard reestablishes old-proc as the current exception handler. If control leaves or subsequently reenters the body via the invocation of a continuation obtained via call/cc, the procedure that was the current exception handler when the continuation was captured is reinstated.

The procedure procedure established by guard binds var to the value it receives and, within the scope of that binding, processes the clauses \( \text{clause}_1 \ \text{clause}_2 \ldots \) in turn, as if contained within an implicit cond expression. This implicit cond expression is evaluated in the continuation of the guard expression, with old-proc as the current exception handler.

If no else clause is provided, guard supplies one that reraises the exception with the same value, as if with raise-continuable, in the continuation of the call to procedure, with old-proc as the current exception handler.

\[
\begin{align*}
\text{(guard (x [else x]) (raise "oops")) } & \Rightarrow "oops" \\
\text{(guard (x [#f #f]) (raise (make-error))) } & \Rightarrow \text{error} \\
\text{(define-syntax try} \\
\text{ (syntax-rules ()} \\
\text{ [(_ el e2 ...) \\
\text{ (guard (x [(error? x) #f]) el e2 ...)])])} \\
\text{(define open-one} \\
\text{ (lambda fn*} \\
\text{ (let loop ([ls fn*])} \\
\text{ (if (null? ls) \\
\text{ (error 'open-one "all open attempts failed" fn*)} \\
\text{ (or (try (open-input-file (car ls)))} \\
\text{ (loop (cdr ls)))))))) \\
\text{; say bar.ss exists but not foo.ss:} \\
\text{(open-one "foo.ss" "bar.ss") } & \Rightarrow \#<\text{input port bar.ss}>
\end{align*}
\]

### Section 11.2. Defining Condition Types

While a program may pass raise or raise-continuable any Scheme value, the best way to describe an exceptional situation is usually to create and pass a condition object. Where the Revised\(^6\) Report requires the implementation to raise exceptions, the value passed to the current exception handler is always a condition object of one or more of the standard condition types described in Section 11.3. User code may create a condition object that is an instance of one or more standard condition types or it may create an extended condition type and create a condition object of that type.

Condition types are similar to record types but are more flexible in that a condition object may be an instance of two or more condition types, even if neither is a subtype of the other. When a condition is an instance of multiple types, it is referred to as a compound condition. Compound conditions are useful for communicating multiple pieces of information about an exception to the exception handler. A condition that is not a compound condition is referred to as a simple condition. In most cases, the distinction between the two is unimportant, and a simple condition is treated as if it were a compound condition with itself as its only simple condition.

**syntax:** &condition

**libraries:** (rnrs conditions), (rnrs)

&condition is a record-type name (Chapter 9) and the root of the condition-type hierarchy. All simple condition types are extensions of this type, and all conditions, whether simple or compound, are considered instances of this type.
A condition object is an instance of a subtype of &condition or a compound condition, possibly created by user code with condition.

(condition? 'stable) ⇒ #f  
(condition? (make-error)) ⇒ #t  
(condition? (make-message-condition "oops")) ⇒ #t  
(condition?  
  (condition  
    (make-error)  
    (make-message-condition "no such element"))) ⇒ #t

procedure: (condition condition ...)
returns: a condition, possibly compound
libraries: (rnrs conditions), (rnrs)

condition is used to create condition objects that may consist of multiple simple conditions. Each argument condition may be simple or complex; if simple, it is treated as a compound condition with itself as its only simple condition. The simple conditions of the result condition are the simple conditions of the condition arguments, flattened into a single list and appearing in order, with the simple conditions of the first condition followed by the simple conditions of the second, and so on.

If the list has exactly one element, the result condition may be simple or compound; otherwise it is compound. The distinction between simple and compound conditions is not usually important but can be detected, if define-record-type rather than define-condition-type is used to extend an existing condition type, via the predicate defined by define-record-type.

(condition) ⇒ #<condition>  
(condition  
  (make-error)  
  (make-message-condition "oops")) ⇒ #<condition>

(define-record-type (&xcond make-xcond xcond?) (parent &condition))  
(xcond? (make-xcond)) ⇒ #t  
(xcond? (condition (make-xcond))) ⇒ #t or #f  
(xcond? (condition)) ⇒ #f  
(xcond? (condition (make-error) (make-xcond))) ⇒ #f

procedure: (simple-conditions condition)
returns: a list of the simple conditions of condition
libraries: (rnrs conditions), (rnrs)

(simple-conditions (condition)) ⇒ '()  
(simple-conditions (make-error)) ⇒ (#<condition &error>)  
(simple-conditions (condition (make-error))) ⇒ (#<condition &error>)  
(simple-conditions  
  (condition  
    (make-error)  
    (make-message-condition  
      "oops"))) ⇒ (#<condition &error> #<condition &message>)

The Scheme Programming Language, 4th Edition
Section 11.2. Defining Condition Types 277
(let ([c1 (make-error)]
    [c2 (make-who-condition "f")]
    [c3 (make-message-condition "invalid argument")]
    [c4 (make-message-condition "error occurred while reading from file")]
    [c5 (make-irritants-condition "a.ss")])
(equal? (simple-conditions
    (condition (condition c1 c2) c3)
    (condition c4 (condition c5)))) (list c1 c2 c3 c4 c5))

syntax: (define-condition-type name parent constructor pred field ...)
libraries: (rnrs conditions), (rnrs)

A define-condition-type form is a definition and may appear anywhere other definitions may appear. It is used to define new simple condition types.

The subforms name, parent, constructor, and pred must be identifiers. Each field must be of the form (field-name accessor-name), where field-name and accessor-name are identifiers.

define-condition-type defines name as a new record type whose parent record type is parent, whose constructor name is constructor, whose predicate name is pred, whose fields are field-name ..., and whose field accessors are named by accessor-name ....

With the exception of the predicate and field accessors, define-condition-type is essentially an ordinary record definition equivalent to

(define-record-type (name constructor pred)
    (parent parent)
    (fields ((immutable field-name accessor-name) ...)))

The predicate differs from one that would be generated by a define-record-type form in that it returns #t not only for an instance of the new type but also for compound conditions whose simple conditions include an instance of the new type. Similarly, field accessors accept instances of the new type as well as compound conditions whose simple conditions include at least one instance of the new record type. If an accessor receives a compound condition whose simple conditions list includes one or more instances of the new type, the accessor operates on the first instance in the list.

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

(mistake? 'booboo) ⇒ #f

(define c1 (make-mistake 'spelling))
(mistake? c1) ⇒ #t
(mistake-type c1) ⇒ spelling

(define c2 (condition c1 (make-irritants-condition '(eggregius))))
(mistake? c2) ⇒ #t
(mistake-type c2) ⇒ spelling
(irritants-condition? c2) ⇒ #t
(condition-irritants c2) ⇒ (eggregius)
procedure: (condition-predicate rtd)
returns: a condition predicate

procedure: (condition-accessor rtd procedure)
returns: a condition accessor

libraries: (rnrs conditions), (rnrs)

These procedures may be used to create the same kind of special predicates and accessors that are created by define-record-type from a record-type descriptor, rtd, of a simple condition type or other type derived from a simple condition type.

For both procedures, rtd must be a record-type descriptor of a subtype of &condition, and for condition-accessor, procedure should accept one argument.

The predicate returned by condition-predicate accepts one argument, which may be any Scheme value. The predicate returns #t if the value is a condition of the type described by rtd, i.e., an instance of the type described by rtd (or one of its subtypes) or a compound condition whose simple conditions include an instance of the type described by rtd. Otherwise, the predicate returns #f.

The accessor returned by condition-accessor accepts one argument, c, which must be a condition of the type described by rtd. The accessor applies procedure to a single argument, the first element of c's simple condition list that is an instance of the type described by rtd (this is c itself if c is a simple condition), and returns the result of this application. In most situations, procedure is a record accessor for a field of the type described by rtd.

(define-record-type (&mistake make-mistake $mistake?)
  (parent &condition)
  (fields (immutable type $mistake-type)))

; define predicate and accessor as if we'd used define-condition-type
(define rtd (record-type-descriptor &mistake))
(define mistake? (condition-predicate rtd))
(define mistake-type (condition-accessor rtd $mistake-type))

(define c1 (make-mistake 'spelling))
(define c2 (condition c1 (make-irritants-condition '(eggregius))))
(list (mistake? c1) (mistake? c2)) ⇒ (#t #t)
(list ($mistake? c1) ($mistake? c2)) ⇒ (#t #f)
(mistake-type c1) ⇒ spelling
($mistake-type c1) ⇒ spelling
(mistake-type c2) ⇒ spelling
($mistake-type c2) ⇒ violation

Section 11.3. Standard Condition Types

syntax: &serious
procedure: (make-serious-condition)
returns: a condition of type &serious

procedure: (serious-condition? obj)
returns: #t if obj is a condition of type &serious, #f otherwise
libraries: (rnrs conditions), (rnrs)

Conditions of this type indicate situations of a serious nature that, if uncaught, generally result in termination of the program's execution. Conditions of this type typically occur as one of the more specific subtypes &error or &violation. This condition type might be defined as follows.
(define-condition-type &serious &condition
  make-serious-condition serious-condition?)

**Syntax:** &violation
**Procedure:** (make-violation)
**Returns:** a condition of type &violation
**Procedure:** (violation? obj)
**Returns:** #t if obj is a condition of type &violation, #f otherwise
**Libraries:** (rnrs conditions), (rnrs)

Conditions of this type indicate that the program has violated some requirement, usually due to a bug in the program. This condition type might be defined as follows.

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

**Syntax:** &assertion
**Procedure:** (make-assertion-violation)
**Returns:** a condition of type &assertion
**Procedure:** (assertion-violation? obj)
**Returns:** #t if obj is a condition of type &assertion, #f otherwise
**Libraries:** (rnrs conditions), (rnrs)

This condition type indicates a specific violation in which the program has passed the wrong number or types of arguments to a procedure. This condition type might be defined as follows.

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

**Syntax:** &error
**Procedure:** (make-error)
**Returns:** a condition of type &error
**Procedure:** (error? obj)
**Returns:** #t if obj is a condition of type &error, #f otherwise
**Libraries:** (rnrs conditions), (rnrs)

Conditions of this type indicate that an error has occurred with the program's interaction with its operating environment, such as the failure of an attempt to open a file. It is not used to describe situations in which an error in the program has been detected. This condition type might be defined as follows.

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

**Syntax:** &warning
**Procedure:** (make-warning)
**Returns:** a condition of type &warning
**Procedure:** (warning? obj)
**Returns:** #t if obj is a condition of type &warning, #f otherwise
**Libraries:** (rnrs conditions), (rnrs)

Warning conditions indicate situations that do not prevent the program from continuing its execution but, in some cases, might result in a more serious problem at some later point. For example, a compiler might use a condition of this type to indicate that it has processed a call to a standard procedure with the wrong number of arguments; this will not become a serious problem unless the call is actually evaluated at some later point.
This condition type might be defined as follows.

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

**syntax:** &message
**procedure:** (make-message-condition message)
**returns:** a condition of type &message
**procedure:** (message-condition? obj)
**returns:** #t if obj is a condition of type &message, #f otherwise
**procedure:** (condition-message condition)
**returns:** the contents of condition's message field
**libraries:** (rnrs conditions), (rnrs)

Conditions of this type are usually included with a &warning condition or one of the &serious condition subtypes to provide a more specific description of the exceptional situation. The message argument to the constructor may be any Scheme value but is typically a string. This condition type might be defined as follows.

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

**syntax:** &irritants
**procedure:** (make-irritants-condition irritants)
**returns:** a condition of type &irritants
**procedure:** (irritants-condition? obj)
**returns:** #t if obj is a condition of type &irritants, #f otherwise
**procedure:** (condition-irritants condition)
**returns:** the contents of condition's irritants field
**libraries:** (rnrs conditions), (rnrs)

Conditions of this type are usually included with a &message condition to provide information about Scheme values that may have caused or been materially involved in the exceptional situation. For example, if a procedure receives the wrong type of argument, it may raise an exception with a compound condition consisting of an assertion condition, a who condition naming the procedure, a message condition stating that the wrong type of argument was received, and an irritants condition listing the argument. The irritants argument to the constructor should be a list. This condition type might be defined as follows.

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

**syntax:** &who
**procedure:** (make-who-condition who)
**returns:** a condition of type &who
**procedure:** (who-condition? obj)
**returns:** #t if obj is a condition of type &who, #f otherwise
**procedure:** (condition-who condition)
**returns:** the contents of condition's who field
**libraries:** (rnrs conditions), (rnrs)

Conditions of this type are often included with a &message condition to identify the syntactic form or procedure that detected the error. The who argument to the constructor should be a symbol or string. This
condition type might be defined as follows.

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

**syntax:** &non-continuable  
**procedure:** (make-non-continuable-violation)  
**returns:** a condition of type &non-continuable  
**procedure:** (non-continuable-violation? obj)  
**returns:** #t if obj is a condition of type &non-continuable, #f otherwise  
**libraries:** (rnrs conditions), (rnrs)

Conditions of this type indicate that a non-continuable violation has occurred. `raise` raises an exception with this type if the current exception handler returns. This condition type might be defined as follows.

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

**syntax:** &implementation-restriction  
**procedure:** (make-implementation-restriction-violation)  
**returns:** a condition of type &implementation-restriction  
**procedure:** (implementation-restriction-violation? obj)  
**returns:** #t if obj is a condition of type &implementation-restriction, #f otherwise  
**libraries:** (rnrs conditions), (rnrs)

An implementation-restriction condition indicates that the program has attempted to exceed some limitation in the implementation, such as when the value of a fixnum addition operation would result in a number that exceeds the implementation's fixnum range. It does not normally indicate a deficiency in the implementation but rather a mismatch between what the program is attempting to do and what the implementation can support. In many cases, implementation restrictions are dictated by the underlying hardware. This condition type might be defined as follows.

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

**syntax:** &lexical  
**procedure:** (make-lexical-violation)  
**returns:** a condition of type &lexical  
**procedure:** (lexical-violation? obj)  
**returns:** #t if obj is a condition of type &lexical, #f otherwise  
**libraries:** (rnrs conditions), (rnrs)

Conditions of this type indicate that a lexical error has occurred in the parsing of a Scheme program or datum, such as mismatched parentheses or an invalid character appearing within a numeric constant. This condition type might be defined as follows.

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

**syntax:** &syntax  
**procedure:** (make-syntax-violation form subform)

Section 11.3. Standard Condition Types
Conditions of this type indicate that a syntax error has occurred in the parsing of a Scheme program. In most implementations, syntax errors are detected by the macro expander. Each of the form and subform arguments to make-syntax-violation should be a syntax object (Section 8.3) or datum, the former indicating the containing form and the latter indicating the specific subform. For example, if a duplicate formal parameter is found in a lambda expression, form might be the lambda expression and subform might be the duplicated parameter. If there is no need to identify a subform, subform should be #f. This condition type might be defined as follows.

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

An undefined condition indicates an attempt to reference an unbound variable. This condition type might be defined as follows.

(define-condition-type &undefined &violation
    make-undefined-violation undefined-violation?
    (form make-undefined-violation)
    (subform make-undefined-violation-subform))

The next several condition types describe conditions that occur when input or output operations fail in some manner.

(define-condition-type &i/o &error
    make-i/o-error i/o-error?
    (form make-i/o-error)
    (subform make-i/o-error-subform))

A condition of type &i/o indicates that an input/output error of some sort has occurred. Conditions of this type typically occur as one of the more specific subtypes described below. This condition type might be defined as follows.

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

Syntax: &i/o-read
Procedure: (make-i/o-read-error)
returns: a condition of type &i/o-read
procedure: (i/o-read-error? obj)
returns: #t if obj is a condition of type &i/o-read, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

This condition type indicates that an error has occurred while reading from a port. This condition type might be defined as follows.

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

syntax: &i/o-write
procedure: (make-i/o-write-error)
returns: a condition of type &i/o-write
procedure: (i/o-write-error? obj)
returns: #t if obj is a condition of type &i/o-write, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

This condition type indicates that an error has occurred while writing to a port. This condition type might be defined as follows.

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

syntax: &i/o-invalid-position
procedure: (make-i/o-invalid-position-error position)
returns: a condition of type &i/o-invalid-position
procedure: (i/o-invalid-position-error? obj)
returns: #t if obj is a condition of type &i/o-invalid-position, #f otherwise
procedure: (i/o-error-position condition)
returns: the contents of condition's position field
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

This condition type indicates an attempt to set a port's position to a position that is out of range for the underlying file or other object. The position argument to the constructor should be the invalid position. This condition type might be defined as follows.

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

syntax: &i/o-filename
procedure: (make-i/o-filename-error filename)
returns: a condition of type &i/o-filename
procedure: (i/o-filename-error? obj)
returns: #t if obj is a condition of type &i/o-filename, #f otherwise
procedure: (i/o-error-filename condition)
returns: the contents of condition's filename field
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

This condition type indicates an input/output error that occurred while operating on a file. The filename argument to the constructor should be the name of the file. This condition type might be defined as follows.
(define-condition-type &i/o-filename &i/o
  make-i/o-filename-error i/o-filename-error?
  (filename i/o-error-filename))

syntax: &i/o-file-protection
procedure: (make-i/o-file-protection-error filename)
returns: a condition of type &i/o-file-protection
procedure: (i/o-file-protection-error? obj)
returns: #t if obj is a condition of type &i/o-file-protection, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

A condition of this type indicates that an attempt has been made to perform some input/output operation on a
file for which the program does not have the proper permission. This condition type might be defined as
follows.

(make-i/o-file-is-read-only-error filename)

procedure: (make-i/o-file-is-read-only-error filename)
returns: a condition of type &i/o-file-is-read-only
procedure: (i/o-file-is-read-only-error? obj)
returns: #t if obj is a condition of type &i/o-file-is-read-only, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

A condition of this type indicates an attempt to treat as writeable a read-only file. This condition type might
be defined as follows.

(make-i/o-file-already-exists-error filename)

procedure: (make-i/o-file-already-exists-error filename)
returns: a condition of type &i/o-file-already-exists
procedure: (i/o-file-already-exists-error? obj)
returns: #t if obj is a condition of type &i/o-file-already-exists, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

A condition of this type indicates a situation in which an operation on a file failed because the file already
exists, e.g., an attempt is made to open an existing file for output without the no-fail file option. This
condition type might be defined as follows.

(make-i/o-file-does-not-exist-error filename)

procedure: (make-i/o-file-does-not-exist-error filename)
returns: a condition of type &i/o-file-does-not-exist
procedure: (i/o-file-does-not-exist-error? obj)
returns: #t if obj is a condition of type &i/o-file-does-not-exist, #f otherwise
libraries: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)
A condition of this type indicates a situation in which an operation on a file failed because the file does not exist, e.g., an attempt is made to open a nonexistent file for input only. This condition type might be defined as follows.

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

**syntax**: &i/o-port  
**procedure**: (make-i/o-port-error pobj)  
**returns**: a condition of type &i/o-port  
**procedure**: (i/o-port-error? obj)  
**returns**: #t if obj is a condition of type &i/o-port, #f otherwise  
**procedure**: (i/o-error-port condition)  
**returns**: the contents of condition's pobj field  
**libraries**: (rnrs io ports), (rnrs io simple), (rnrs files), (rnrs)

A condition of this type is usually included with a condition of one of the other &i/o subtypes to indicate the port involved in the exceptional situation, if a port is involved. The pobj argument to the constructor should be the port. This condition type might be defined as follows.

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

**syntax**: &i/o-decoding  
**procedure**: (make-i/o-decoding-error pobj)  
**returns**: a condition of type &i/o-decoding  
**procedure**: (i/o-decoding-error? obj)  
**returns**: #t if obj is a condition of type &i/o-decoding, #f otherwise  
**libraries**: (rnrs io ports), (rnrs)

A condition of this type indicates that a decoding error has occurred during the transcoding of bytes to characters. The pobj argument to the constructor should be the port involved, if any, and the port should be positioned past the invalid encoding. This condition type might be defined as follows.

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

**syntax**: &i/o-encoding  
**procedure**: (make-i/o-encoding-error pobj cobj)  
**returns**: a condition of type &i/o-encoding  
**procedure**: (i/o-encoding-error? obj)  
**returns**: #t if obj is a condition of type &i/o-encoding, #f otherwise  
**procedure**: (i/o-encoding-error-char condition)  
**returns**: the contents of condition's cobj field  
**libraries**: (rnrs io ports), (rnrs)

A condition of this type indicates that an encoding error has occurred during the transcoding of characters to bytes. The pobj argument to the constructor should be the port involved, if any, and the cobj argument should be the character for which the encoding failed. This condition type might be defined as follows.

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

A condition of this type indicates a situation in which an operation on a file failed because the file does not exist, e.g., an attempt is made to open a nonexistent file for input only. This condition type might be defined as follows.

```
(define-condition-type &i/o-file-does-not-exist &i/o-filename
  make-i/o-file-does-not-exist-error
  i/o-file-does-not-exist-error?)
```
The final two condition types describe conditions that occur when implementations are required to produce a NaN or infinity but have no representations for these values.

**syntax:** `&no-infinities`

**procedure:** `(make-no-infinities-violation)`

**returns:** a condition of type `&no-infinities`

**procedure:** `(no-infinities-violation? obj)`

**returns:** `#t` if `obj` is a condition of type `&no-infinities`, `#f` otherwise

**libraries:** `(rnrs arithmetic flonums), (rnrs)`

This condition indicates that the implementation has no representation for infinity. This condition type might be defined as follows.

```
(define-condition-type &no-infinities &implementation-restriction
  make-no-infinities-violation
  no-infinities-violation?)
```

**syntax:** `&no-nans`

**procedure:** `(make-no-nans-violation)`

**returns:** a condition of type `&no-nans`

**procedure:** `(no-nans-violation? obj)`

**returns:** `#t` if `obj` is a condition of type `&no-nans`, `#f` otherwise

**libraries:** `(rnrs arithmetic flonums), (rnrs)`

This condition indicates that the implementation has no representation for NaN. This condition type might be defined as follows.

```
(define-condition-type &no-nans &implementation-restriction
  make-no-nans-violation
  no-nans-violation?)
```
Chapter 12. Extended Examples

This chapter presents a series of programs that perform more complicated tasks than most of the examples found throughout the earlier chapters of the book. They illustrate a variety of programming techniques and demonstrate a particular programming style.

Each section of this chapter describes one program in detail and gives examples of its use. This is followed by a listing of the code. At the end of each section are exercises intended to stimulate thought about the program and to suggest possible extensions. These exercises are generally more difficult than those found in Chapters 2 and 3, and a few are major projects.

Section 12.1 presents a simple matrix multiplication package. It demonstrates a set of procedures that could be written in almost any language. Its most interesting features are that all multiplication operations are performed by calling a single generic procedure, \texttt{mul}, which calls the appropriate help procedure depending upon the dimensions of its arguments, and that it dynamically allocates results of the proper size. Section 12.2 presents a merge sorting algorithm for ordering lists according to arbitrary predicates. Section 12.3 describes a syntactic form that is used to construct sets. It demonstrates a simple but efficient syntactic transformation from set notation to Scheme code. Section 12.4 presents a word-counting program borrowed from \textit{The C Programming Language} [19], translated from C into Scheme. It shows character and string manipulation, data structure creation and manipulation, and basic file input and output. Section 12.5 presents a Scheme printer that implements basic versions of \texttt{put-datum}, \texttt{write}, and \texttt{display}. Section 12.6 presents a simple formatted output facility similar to those found in many Scheme systems and in other languages. Section 12.7 presents a simple interpreter for Scheme that illustrates Scheme as a language implementation vehicle while giving an informal operational semantics for Scheme as well as a useful basis for investigating extensions to Scheme. Section 12.8 presents a small, extensible abstract object facility that could serve as the basis for an entire object-oriented subsystem. Section 12.9 presents a recursive algorithm for computing the Fourier transform of a sequence of input values. It highlights the use of Scheme’s complex arithmetic. Section 12.10 presents a concise unification algorithm that shows how procedures can be used as continuations and as substitutions (unifiers) in Scheme. Section 12.11 describes a multitasking facility and its implementation in terms of continuations.

Section 12.1. Matrix and Vector Multiplication

This example program involves mostly basic programming techniques. It demonstrates simple arithmetic and vector operations, looping with the \texttt{do} syntactic form, dispatching based on object type, and raising exceptions.

Multiplication of scalar to scalar, scalar to matrix, or matrix to matrix is performed by a single generic procedure, \texttt{mul}. \texttt{mul} is called with two arguments, and it decides based on the types of its arguments what operation to perform. Because scalar operations use Scheme’s multiplication procedure, \texttt{*}, \texttt{mul} scalars can be any built-in numeric type (exact or inexact complex, real, rational, or integer).

The product of an \( m \times n \) matrix \( A \) and an \( n \times p \) matrix \( B \) is the \( m \times p \) matrix \( C \) whose entries are defined by

\[
C_{ij} = \sum_{k=0}^{n-1} A_{ik}B_{kj},
\]

The product of a scalar \( x \) and an \( m \times n \) matrix \( A \) is the \( m \times n \) matrix \( C \) whose entries are defined by the equation

\[
C_{ij} = xA_{ij},
\]

That is, each element of \( C \) is the product of \( x \) and the corresponding element of \( A \). Vector-vector, vector-matrix, and matrix-vector multiplication may be considered special cases of matrix-matrix multiplication, where a vector is represented as a \( 1 \times n \) or \( n \times 1 \) matrix.
Here are a few examples, each preceded by the equivalent operation in standard mathematical notation.

- **Scalar times scalar:**
  
  \[3 \times 4 = 12\]
  
  \[(\text{mul} \ 3 \ 4) \Rightarrow 12\]

- **Scalar times vector (1 \times 3 matrix):**
  
  \[\frac{1}{2} \times \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 1 & 3/2 \end{pmatrix}\]
  
  \[(\text{mul} \ \frac{1}{2} \ \#'((1 \ 2 \ 3))) \Rightarrow #'((1/2 \ 1 \ 3/2))\]

- **Scalar times matrix:**
  
  \[-2 \times \begin{pmatrix} 3 & -2 & -1 \\ -3 & 0 & -\frac{6}{7} \\ 7 & -1 & -1 \end{pmatrix} = \begin{pmatrix} -6 & 4 & 2 \\ 6 & 0 & 10 \\ -14 & 2 & 2 \end{pmatrix}\]
  
  \[(\text{mul} \ -2 \ \#'((3 \ -2 \ -1) \\ #(-3 \ 0 \ -5) \\ #(-7 \ -1 \ -1))) \Rightarrow #'((-6 \ 4 \ 2) \\ #(-6 \ 0 \ 10) \\ #(-14 \ 2 \ 2))\]

- **Vector times matrix:**
  
  \[\begin{pmatrix} 1 & 3 & 3 \end{pmatrix} \times \begin{pmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{pmatrix} = \begin{pmatrix} 13 & 26 \end{pmatrix}\]
  
  \[(\text{mul} \ \#'((1 \ 2 \ 3)) \\ #'((2 \ 3) \\ #(-3 \ 4) \\ #(-4 \ 5))) \Rightarrow #'((20 \ 26))\]

- **Matrix times vector:**
  
  \[\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 20 \\ 26 \end{pmatrix}\]
  
  \[(\text{mul} \ \#'((2 \ 3 \ 4) \\ #(-3 \ 4 \ 5) \\ #'((1) \ #(-2) \ #(3))) \Rightarrow #'((-20) \ #(-26))\]

- **Matrix times matrix:**
  
  \[\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 5 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 14 & 20 & 26 & 32 \\ 22 & 47 & 62 & 77 \end{pmatrix}\]
  
  \[(\text{mul} \ \#'((1 \ 2 \ 3) \\ #(-4 \ 5 \ 6) \\ #'((1 \ 2 \ 3) \\ #(-2 \ 3 \ 4) \\ #(-3 \ 4 \ 5) \\ #(-3 \ 4 \ 5 \ 6))) \Rightarrow #'((-14 \ 20 \ 26 \ 32) \\ #(-32 \ 47 \ 62 \ 77))\]
The code for `mul` and its helpers, which is structured as a library, appears below. The first few definitions establish a set of procedures that support the matrix data type. A matrix is a vector of vectors. Included are a procedure to create matrices, procedures to access and assign matrix elements, and a matrix predicate. Following these definitions is the definition of `mul` itself. Inside the lambda expression for `mul` are a set of definitions for help procedures that support `mul`.

`mul` checks the types of its arguments and chooses the appropriate help procedure to do the work. Each helper operates on arguments of specific types. For example, `mat-sca-mul` multiplies a matrix by a scalar. If the type of either argument is invalid or the arguments are incompatible, e.g., rows or columns do not match up, `mul` or one of its helpers raises an exception.

(library (tspl matrix)
  (export make-matrix matrix? matrix-rows matrix-columns
       matrix-ref matrix-set! mul)
  (import (rnrs))
  ; make-matrix creates a matrix (a vector of vectors).
  (define make-matrix
    (lambda (rows columns)
      (do ([m (make-vector rows)]
           [i 0 (+ i 1)])
         ((= i rows) m)
       (vector-set! m i (make-vector columns)))))

  ; matrix? checks to see if its argument is a matrix.
  ; It isn't foolproof, but it's generally good enough.
  (define matrix?
    (lambda (x)
      (and (vector? x)
           (> (vector-length x) 0)
           (vector? (vector-ref x 0)))))

  ; matrix-rows returns the number of rows in a matrix.
  (define matrix-rows
    (lambda (x)
      (vector-length x)))

  ; matrix-columns returns the number of columns in a matrix.
  (define matrix-columns
    (lambda (x)
      (vector-length (vector-ref x 0)))))

  ; matrix-ref returns the jth element of the ith row.
  (define matrix-ref
    (lambda (m i j)
      (vector-ref (vector-ref m i) j)))

  ; matrix-set! changes the jth element of the ith row.
  (define matrix-set!
    (lambda (m i j x)
      (vector-set! (vector-ref m i) j x)))

  ; mat-sca-mul multiplies a matrix by a scalar.
  (define mat-sca-mul
(lambda (m x)
  (let* ([nr (matrix-rows m)]
         [nc (matrix-columns m)]
         [r (make-matrix nr nc)])
    (do ([i 0 (+ i 1)])
        (if (= i nr) r)
    (do ([j 0 (+ j 1)])
        (if (= j nc)
            (matrix-set! r i j (* x (matrix-ref m i j)))))))

; mat-mat-mul multiplies one matrix by another, after verifying
; that the first matrix has as many columns as the second
; matrix has rows.
(define mat-mat-mul
  (lambda (m1 m2)
    (let* ([nr1 (matrix-rows m1)]
           [nr2 (matrix-rows m2)]
           [nc2 (matrix-columns m2)]
           [r (make-matrix nr1 nc2)])
      (unless (= (matrix-columns m1) nr2) (match-error m1 m2))
      (do ([i 0 (+ i 1)])
          (if (= i nr1) r)
      (do ([j 0 (+ j 1)])
          (if (= j nc2)
              (do ([k 0 (+ k 1)]
                  [a 0 (+ a
                    (* (matrix-ref m1 i k) (matrix-ref m2 k j)))]
                  (if (= k nr2)
                      (matrix-set! r i j a))))))))

; type-error is called to complain when mul receives an invalid
; type of argument.
(define type-error
  (lambda (what)
    (assertion-violation 'mul
      "not a number or matrix"
      what)))

; match-error is called to complain when mul receives a pair of
; incompatible arguments.
(define match-error
  (lambda (what1 what2)
    (assertion-violation 'mul
      "incompatible operands" what1
      what2)))

; mul is the generic matrix/scalar multiplication procedure
(define mul
  (lambda (x y)
    (cond
      [(number? x)
        (cond
          [(number? y) (* x y)]
          ...]]

Section 12.1. Matrix and Vector Multiplication
[(matrix? y) (mat-sca-mul y x)]
[else (type-error y)]))
[(matrix? x)
 (cond
  [(number? y) (mat-sca-mul x y)]
  [(matrix? y) (mat-mat-mul x y)]
  [else (type-error y)]))
[else (type-error x)])])

Exercise 12.1.1

Make the necessary changes to rename mul to *.

Exercise 12.1.2

The predicate matrix? is usually sufficient but not completely reliable, since it may return #t for objects that are not matrices. In particular, it does not verify that all of the matrix rows are vectors, that each row has the same number of elements, or that the elements themselves are numbers. Modify matrix? to perform each of these additional checks.

Exercise 12.1.3

Another solution to Exercise 12.1.2 is to define a matrix record type encapsulating the vectors of the matrix. If the matrix creation routine never allows a malformed matrix record to be created, a matrix record check is the only check needed to ensure that the input is well formed. Define a matrix record type and recode the library to use it.

Exercise 12.1.4

Write similar generic procedures for addition and subtraction. Devise a generic dispatch procedure or syntactic form so that the type dispatching code need not be rewritten for each new operation.

Exercise 12.1.5

This version of mul uses vectors of vectors to represent matrices. Rewrite the system, using nested lists to represent matrices. What efficiency is gained or lost by this change?

Section 12.2. Sorting

This section illustrates a list sorting algorithm based on a simple technique known as merge sorting. The procedure sort defined here accepts two arguments: a predicate and a list, just like the built-in list-sort. It returns a list containing the elements of the old list sorted according to the predicate. As with list-sort, the predicate should be a procedure that expects two arguments and returns #t if its first argument must precede its second in the sorted list and false otherwise. That is, if the predicate is applied to two elements x and y, where x appears after y in the input list, it should return true only if x should appear before y in the output list. If this constraint is met, sort will perform a stable sort; with a stable sort, two elements that are already sorted with respect to each other will appear in the output in the same order in which they appeared in the input. Thus, sorting a list that is already sorted will result in no reordering, even if there are equivalent elements.

(sort < '(3 4 2 1 2 5)) ⇒ (1 2 2 3 4 5)
(sort > '(0.5 1/2)) ⇒ (0.5 1/2)
(sort > '(1/2 0.5)) ⇒ (1/2 0.5)
(list->string
A companion procedure, \texttt{merge}, is also defined by the code. \texttt{merge} accepts a predicate and two sorted lists and returns a merged list in sorted order of the elements of the two lists. With a properly defined predicate, \texttt{merge} is also stable in the sense that an item from the first list will appear before an item from the second list unless it is necessary that the item from the second list appear first.

\begin{verbatim}
(merge char>?
  '(#\a #\c)
  '(#\b #\c #\d)) ⇒ (#\a #\b #\c #\c #\d)
(merge <
  '(1/2 2/3 3/4)
  '(0.5 0.6 0.7)) ⇒ (1/2 0.5 0.6 2/3 0.7 3/4)
(list->string
  (merge char>?
    (string->list "old")
    (string->list "toe"))) ⇒ "tooled"
\end{verbatim}

The merge sorting algorithm is simple and elegant. The input list is split into two approximately equal sublists. These sublists are sorted recursively, yielding two sorted lists. The sorted lists are then merged to form a single sorted list. The base case for the recursion is a list of one element, which is already sorted.

To reduce overhead, the implementation computes the length of the input list once, in \texttt{sort}, rather than at each step of the recursion, in \texttt{dosort}. This also allows \texttt{dosort} to isolate the first half of the list merely by halving the length, saving the cost of allocating a new list containing half of the elements. As a result, \texttt{ls} may contain more than \texttt{n} elements, but only the first \texttt{n} elements are considered part of the list.

\begin{verbatim}
(library (tspl sort)
  (export sort merge)
  (import (rnrs))

  (define dosort
    (lambda (pred? ls n)
      (if (= n 1)
        (list (car ls))
        (let ([i (div n 2)])
          (domerge pred? (dosort pred? ls i)
            (dosort pred? (list-tail ls i) (- n i)))))))

  (define domerge
    (lambda (pred? l1 l2)
      (cond
        [[null? l1] l2]
        [[null? l2] l1]
        [(pred? (car l2) (car l1))
          (cons (car l2) (domerge pred? l1 (cdr l2)))]
        [else (cons (car l1) (domerge pred? (cdr l1) l2))])))

  (define sort
    (lambda (pred? l)
      (if (null? l) l (dosort pred? l (length l)))))
\end{verbatim}

The Scheme Programming Language, 4th Edition

294

Section 12.2. Sorting
(define merge
  (lambda (pred? l1 l2)
    (domerge pred? l1 l2))))

Exercise 12.2.1

In dosort, when n is 1, why is (list (car ls)) returned instead of just ls? How much allocation would be saved overall by replacing (list (car ls)) with
(if (null? (cdr ls)) ls (list (car ls)))?

Exercise 12.2.2

How much work is actually saved by not copying the first part of the input list when splitting it in dosort?

Exercise 12.2.3

All or nearly all allocation could be saved if the algorithm were to work destructively, using set-cdr! to separate and join lists. Write destructive versions sort! and merge! of the sort and merge. Determine the difference between the two sets of procedures in terms of allocation and run time for various inputs.

Section 12.3. A Set Constructor

This example describes a syntactic extension, set-of, that allows the construction of sets represented as lists with no repeated elements [22]. It uses define-syntax and syntax-rules to compile set expressions into recursion expressions. The expanded code is often as efficient as that which can be produced by hand.

A set-of expression takes the following form.

(set-of expr clause ...)

expr describes the elements of the set in terms of the bindings established by the set-of clauses
clause .... Each clause can take one of three forms:

1. A clause of the form (x in s) establishes a binding for x to each element of the set s in turn. This binding is visible within the remaining clauses and the expression expr.

2. A clause of the form (x is e) establishes a binding for x to e. This binding is visible within the remaining clauses and the expression expr. This form is essentially an abbreviation for (x in (list e)).

3. A clause taking any other form is treated as a predicate; this is used to force refusal of certain elements as in the second of the examples below.

(set-of x
  (x in '(a b c))) ⇒ (a b c)

(set-of x
  (x in '(1 2 3 4))
  (even? x)) ⇒ (2 4)

(set-of (cons x y)
  (x in '(1 2 3))
  (y is (* x x))) ⇒ (((1 . 1) (2 . 4) (3 . 9)))
A set-of expression is transformed into nested let, named let, and if expressions, corresponding to each is, in, or predicate subexpression. For example, the simple expression

(set-of x (x in '(a b c)))

is transformed into

(let loop ([set '(a b c)])
  (if (null? set)
      '()
      (let ([x (car set)])
        (set-cons x (loop (cdr set))))))

The expression

(set-of x (x in '(1 2 3 4)) (even? x))

is transformed into

(let loop ([set '(1 2 3 4)])
  (if (null? set)
      '()
      (let ([x (car set)])
        (if (even? x)
            (set-cons x (loop (cdr set)))
            (loop (cdr set))))))

The more complicated expression

(set-of (cons x y) (x in '(1 2 3)) (y is (* x x)))

is transformed into

(let loop ([set '(1 2 3)])
  (if (null? set)
      '()
      (let ([x (car set)])
        (let ([y (* x x)])
          (set-cons (cons x y)
                    (loop (cdr set)))))))

Finally, the expression

(set-of (cons x y) (x in '(a b)) (y in '(1 2)))

is transformed into nested named let expressions:

(let loop1 ([set1 '(a b)])
  (if (null? set1)
      '()
      (let ([x (car set1)]
            [y (* x x)])
        (set-cons (cons x y)
                  (loop (cdr set1)))))

The Scheme Programming Language, 4th Edition

Section 12.3. A Set Constructor
These are fairly straightforward transformations, except that the base case for the recursion on nested named let expressions varies depending upon the level. The base case for the outermost named let is always the empty list (), while the base case for an internal named let is the recursion step for the next outer named let. In order to handle this, the definition of set-of employs a help syntactic extension set-of-help. set-of-help takes an additional expression, base, which is the base case for recursion at the current level.

(define-syntax set-of
  (syntax-rules ()
    [(_ e m ...) (set-of-help e '() m ...)]
    [(_ e base (x in s) m ...) (let loop ([set s])
      (if (null? set)
        base
        (let ([x (car set)])
          (set-of-help e (loop (cdr set)) m ...))))]
    [(_ e base (x is y) m ...) (let ([x y]) (set-of-help e base m ...))]
    [(_ e base p m ...) (if p (set-of-help e base m ...) base)])

; set-of-help recognizes in, is, and predicate expressions and
; changes them into nested named let, let, and if expressions.
(define-syntax set-of-help
  (syntax-rules (in is)
    [(_ e base) (set-cons e base)]
    [(_ e base (x in s) m ...) (let loop ([set s])
      (if (null? set)
        base
        (let ([x (car set)])
          (set-of-help e (loop (cdr set)) m ...))))]
    [(_ e base (x is y) m ...) (let ([x y]) (set-of-help e base m ...))]
    [(_ e base p m ...) (if p (set-of-help e base m ...) base)])

; since in and is are used as auxiliary keywords by set-of, the
; library must export definitions for them as well
(define-syntax in
  (lambda (x)
    (syntax-violation 'in "misplaced auxiliary keyword" x)))

(define-syntax is
  (lambda (x)
    (syntax-violation 'is "misplaced auxiliary keyword" x)))

; set-cons returns the original set y if x is already in y.
(define set-cons

Section 12.3. A Set Constructor
Exercise 12.3.1

Write a procedure, `union`, that takes an arbitrary number of sets (lists) as arguments and returns the union of the sets, using only the `set-of` syntactic form. For example:

```
    (union) ➞ ()
    (union '(a b c)) ➞ (a b c)
    (union '(2 5 4) '(9 4 3)) ➞ (2 5 9 4 3)
    (union '(1 2) '(2 4) '(4 8)) ➞ (1 2 4 8)
```

Exercise 12.3.2

A single-list version of `map` can (almost) be defined as follows.

```
    (define map1
      (lambda (f ls)
        (set-of (f x) (x in ls))))

    (map1 - '(1 2 3 2)) ➞ (-1 -3 -2)
```

Why does this not work? What could be changed to make it work?

Exercise 12.3.3

Devise a different definition of `set-cons` that maintains sets in some sorted order, making the test for set membership, and hence `set-cons` itself, potentially more efficient.

Section 12.4. Word Frequency Counting

This program demonstrates several basic programming techniques, including string and character manipulation, file input/output, data structure manipulation, and recursion. The program is adapted from Chapter 6 of *The C Programming Language* [19]. One reason for using this particular example is to show how a C program might look when converted almost literally into Scheme.

A few differences between the Scheme program and the original C program are worth noting. First, the Scheme version employs a different protocol for file input and output. Rather than implicitly using the standard input and output ports, it requires that filenames be passed in, thus demonstrating the opening and closing of files. Second, the procedure `get-word` returns one of three values: a string (the word), a nonalphabetic character, or an eof value. The original C version returned a flag for letter (to say that a word was read) or a nonalphabetic character. Furthermore, the C version passed in a string to fill and a limit on the number of characters in the string; the Scheme version builds a new string of whatever length is required (the characters in the word are held in a list until the end of the word has been found, then converted into a string with `list->string`). Finally, `char-type` uses the primitive Scheme character predicates `char-alphabetic?` and `char-numeric?` to determine whether a character is a letter or digit.

The main program, `frequency`, takes an input filename and an output filename as arguments, e.g.,

```
    (frequency "pickle" "freq.out")
```

prints the frequency count for each word in the file "pickle" to the file "freq.out." As `frequency` reads words from the input file, it inserts them into a binary tree structure (using a binary sorting algorithm). Duplicate entries are recorded by incrementing the count associated with
each word. Once end of file is reached, the program traverses the tree, printing each word with its count.

Assume that the file "pickle" contains the following text.

Peter Piper picked a peck of pickled peppers;
A peck of pickled peppers Peter Piper picked.
If Peter Piper picked a peck of pickled peppers,
Where's the peck of pickled peppers Peter Piper picked?

Then, after typing `(frequency "pickle" "freq.out")`, the file "freq.out" should contain the following.

```
1 A
1 If
4 Peter
4 Piper
1 Where
2 a
4 of
4 peck
4 peppers
4 picked
4 pickled
1 s
1 the
```

The code for the word-counting program is structured as a top-level program, with the script header recommended in the scripts chapter of the nonnormative appendices to the Revised\(^6\) Report [25]. It takes the names of input and output files from the command line.

```
#!/usr/bin/env scheme-script
(import (rnrs))

;;; If the next character on p is a letter, get-word reads a word
;;; from p and returns it in a string. If the character is not a
;;; letter, get-word returns the character (on eof, the eof-object).
(define get-word
  (lambda (p)
    (let ([c (get-char p)])
      (if (eq? (char-type c) 'letter)
          (list->string
            (let loop ([c c])
              (cons
                c
                (if (memq (char-type (lookahead-char p))
                    '(letter digit))
                  (loop (get-char p))
                  '()))))
          c)))))

;;; char-type tests for the eof-object first, since the eof-object
;;; may not be a valid argument to char-alphabetic? or char-numeric?
;;; It returns the eof-object, the symbol letter, the symbol digit,
;;; or the argument itself if it is not a letter or digit.

Section 12.4. Word Frequency Counting

299
;;; Tree nodes are represented as a record type with four fields: word, left, right, and count. Only one field, word, is initialized by an argument to the constructor procedure make-tnode. The remaining fields are initialized by the constructor and changed by subsequent operations.
(define-record-type tnode
  (fields (immutable word)
    (mutable left)
    (mutable right)
    (mutable count))
  (protocol
    (lambda (new)
      (lambda (word)
        (new word '() '() 1)))))

;;; If the word already exists in the tree, tree increments its count. Otherwise, a new tree node is created and put into the tree. In any case, the new or modified tree is returned.
(define tree
  (lambda (node word)
    (cond
      [(null? node) (make-tnode word)]
      [(string=? word (tnode-word node))
        (tnode-count-set! node (+ (tnode-count node) 1))
        node]
      [(string<? word (tnode-word node))
        (tnode-left-set! node (tree (tnode-left node) word))
        node]
      [else
        (tnode-right-set! node (tree (tnode-right node) word))
        node]]))

;;; tree-print prints the tree in "in-order," i.e., left subtree, then node, then right subtree. For each word, the count and the word are printed on a single line.
(define tree-print
  (lambda (node p)
    (unless (null? node)
      (tree-print (tnode-left node) p)
      (put-datum p (tnode-count node))
      (put-char p #\space)
      (put-string p (tnode-word node))
      (newline p)
      (tree-print (tnode-right node) p))))
define frequency
(lambda (infn outfn)
  (let ((ip (open-file-input-port infn (file-options)
       (buffer-mode block) (native-transcoder)))
      [op (open-file-output-port outfn (file-options)
       (buffer-mode block) (native-transcoder))])
    (let loop ([root '()])
      (let ([w (get-word ip)])
        (cond
          [(eof-object? w) (tree-print root op)]
          [(string? w) (loop (tree root w))]
          [else (loop root)])
        (close-port ip)
        (close-port op))))

(unless (= (length (command-line)) 3)
  (put-string (current-error-port) "usage: ")
  (put-string (current-error-port) (car (command-line)))
  (put-string (current-error-port) " input-filename output-filename\n")
  (exit #f))

(frequency (cadr (command-line)) (caddr (command-line)))

Exercise 12.4.1

In the output file shown earlier, the capitalized words appeared before the others in the output file, and the capital A was not recognized as the same word as the lower-case a. Modify tree to use the case-insensitive versions of the string comparisons so that this does not happen.

Exercise 12.4.2

The "word" s appears in the file "freq.out," although it is really just a part of the contraction Where's. Adjust get-word to allow embedded single quote marks.

Exercise 12.4.3

Modify this program to "weed out" certain common words such as a, an, the, is, of, etc., in order to reduce the amount of output for long input files. Try to devise other ways to cut down on useless output.

Exercise 12.4.4

get-word buffers characters in a list, allocating a new pair (with cons) for each character. Make it more efficient by using a string to buffer the characters. Devise a way to allow the string to grow if necessary. [Hint: Use string-append or a string output port.]

Exercise 12.4.5

The tree implementation works by creating trees and later filling in their left and right fields. This requires many unnecessary assignments. Rewrite the tree procedure to avoid tree-left-set! and tree-right-set! entirely.
Exercise 12.4.6

Recode the program to use a hashtable (Section 6.13) in place of a binary tree, and compare the running times of the new and old programs on large input files. Are hashtables always faster or always slower? Is there a break-even point? Does the break-even point depend on the size of the file or on some other characteristic of the file?

Section 12.5. Scheme Printer

Printing Scheme objects may seem like a complicated process, but in fact a rudimentary printer is straightforward, as this example demonstrates. put-datum, write, and display are all implemented by the same code. Sophisticated printers often support various printer controls and handle printing of cyclic objects, but the one given here is completely basic.

The main driver for the program is a procedure \texttt{wr}, which takes an object to print \texttt{x}, a flag \texttt{d?}, and a port \texttt{p}. The flag \texttt{d?} (for display) is \texttt{#t} if the code is to \textit{display} the object, \texttt{#f} otherwise. The \texttt{d?} flag is important only for characters and strings. Recall from Section 7.8 that \textit{display} prints strings without the enclosing quote marks and characters without the the \texttt{"\textbackslash\} syntax.

The entry points for \texttt{write} and \texttt{display} handle the optionality of the second (port) argument, passing the value of \texttt{current-output-port} when no port argument is provided.

Procedures, ports, and the end-of-file object are printed as \texttt{#<procedure>}, \texttt{#<port>}, and \texttt{#<eof>}. Unrecognized types of values are printed as \texttt{#<unknown>}. So, for example, a hashtable, enumeration set, and object of some implementation-specific type will all print as \texttt{#<unknown>}.

```
(library (tspl printer)
  (export put-datum write display)
  (import (except (rnrs) put-datum write display))

; define these here to avoid confusing paren-balancers
(define lparen #\()
(define rparen #\))

; \texttt{wr} is the driver, dispatching on the type of \texttt{x}
(define wr
  (lambda (x d? p)
    (cond
      [(symbol? x) (put-string p (symbol->string x))]
      [(pair? x) (wrpair x d? p)]
      [(number? x) (put-string p (number->string x))]
      [(null? x) (put-string p "()")]
      [(boolean? x) (put-string p (if x="#t"="#f"))]]
      [(char? x) (if d? (put-char p x) (wrchar x p))]
      [(string? x) (if d? (put-string p x) (wrstring x p))]
      [(vector? x) (wrvector x d? p)]
      [(bytevector? x) (wrbytevector x d? p)]
      [(eof-object? x) (put-string p "#<eof>")]
      [(port? x) (put-string p "#<port>")]
      [(procedure? x) (put-string p "#<procedure>")]])
)

; \texttt{wrpair} handles pairs and nonempty lists
(define wrpair

302 Section 12.4. Word Frequency Counting
(lambda (x d? p)
  (put-char p lparen)
  (let loop ([x x])
    (wr (car x) d? p)
    (cond
      [(pair? (cdr x)) (put-char p \space) (loop (cdr x))]
      [(null? (cdr x))]
      [else (put-string p " . ") (wr (cdr x) d? p)])
    (put-char p rparen)))

; wrchar handles characters. Used only when d? is #f.
(define wrchar
  (lambda (x p)
    (put-string p "\""
    (cond
      [(assq x '(({\alarm . "alarm") (\backspace . "backspace")
                   (\delete . "delete") (\esc . "esc")
                   (\newline . "newline") (\nul . "nul")
                   (\page . "page") (\return . "return")
                   (\space . "space") (\tab . "tab")
                   (\vtab . "vtab"))) =>
        (lambda (a) (put-string p (cdr a)))
      [else (put-char p x)]))))

; wrstring handles strings. Used only when d? is #f.
(define wrstring
  (lambda (x p)
    (put-char p #")
    (let ([n (string-length x)])
      (do ([i 0 (+ i 1)]
           (\eq i n))
        (let ([c (string-ref x i)])
          (case c
            [(\alarm) (put-string p \a)]
            [(\backspace) (put-string p \b)]
            [(\newline) (put-string p \n)]
            [(\page) (put-string p \f)]
            [(\return) (put-string p \r)]
            [(\tab) (put-string p \t)]
            [(\vtab) (put-string p \v)]
            [(\") (put-string p \"]
            [(\\) (put-string p \")]
            [else (put-char p c)])]))
    (put-char p #")))

(define wrvector
  (lambda (x d? p)
    (put-char p \\#)
    (let ([n (vector-length x)])
      (do ([i 0 (+ i 1)] [sep lparen \space]
           (\eq i n))
        (put-char p sep)
        (wr (vector-ref x i) d? p)))
    (put-char p rparen)))
Exercise 12.5.1

Numbers are printed with the help of `number->string`. Correct printing of all Scheme numeric types, especially inexact numbers, is a complicated task. Handling exact integers and ratios is fairly straightforward, however. Modify the code to print exact integers and ratios numbers directly (without `number->string`), but continue to use `number->string` for inexact and complex numbers.

Exercise 12.5.2

Modify `wr` and its helpers to direct their output to an internal buffer rather than to a port. Use the modified versions to implement a procedure `object->string` that, like `number->string`, returns a string containing a printed representation of its input. For example:

```
(object->string '(a b c))  =>  "(a b c)"
(object->string "hello")  =>  ""hello"
```

You may be surprised just how easy this change is to make.
Exercise 12.5.3

Some symbols are not printed properly by \texttt{wr}, including those that start with digits or contain whitespace. Modify \texttt{wr} to call a \texttt{wrsymbol} helper that uses hex scalar escapes as necessary to handle such symbols. A hex scalar escape takes the form \texttt{\#xnn;}, where \texttt{n} is the Unicode scalar value of a character expressed in hexadecimal notation. Consult the grammar for symbols on page 458 to determine when hex scalar escapes are necessary.

Section 12.6. Formatted Output

It is often necessary to print strings containing the printed representations of Scheme objects, especially numbers. Doing so with Scheme’s standard output routines can be tedious. For example, the \texttt{tree-print} procedure of Section 12.4 requires a sequence of four calls to output routines to print a simple one-line message:

\begin{verbatim}
(put-datum p (tnode-count node))
(put-char p #\space)
(put-string p (tnode-word node))
(newline p)
\end{verbatim}

The formatted output facility defined in this section allows these four calls to be replaced by the single call to \texttt{fprintf} below.

\begin{verbatim}
(ftprintf p "\~s \~a\~%" (tnode-count node) (tnode-word node))
\end{verbatim}

\texttt{fprintf} expects a port argument, a \textit{control string}, and an indefinite number of additional arguments that are inserted into the output as specified by the control string. In the example, the value of \texttt{(tnode-count node)} is written first, in place of \texttt{\~s}. This is followed by a space and the displayed value of \texttt{(tnode-word node)}, in place of \texttt{\~a}. The \texttt{\~%} is replaced in the output with a newline.

The procedure \texttt{printf}, also defined in this section, is like \texttt{fprintf} except that no port argument is expected and output is sent to the current output port.

\texttt{\~s}, \texttt{\~a}, and \texttt{\~%} are \textit{format directives}; \texttt{\~s} causes the first unused argument after the control string to be printed to the output via \texttt{write}, \texttt{\~a} causes the first unused argument to be printed via \texttt{display}, and \texttt{\~%} simply causes a newline character to be printed. The simple implementation of \texttt{fprintf} below recognizes only one other format directive, \texttt{\~}, which inserts a tilde into the output. For example,

\begin{verbatim}
(printf "The string \~s displays as \~.\~%" "~")
\end{verbatim}

prints

The string "~" displays as ~.

\begin{verbatim}
(library (tspl formatted-output)
  (export printf fprintf)
  (import (rnrs))

; dofmt does all of the work. It loops through the control string
; recognizing format directives and printing all other characters
; without interpretation. A tilde at the end of a control string is
; treated as an ordinary character. No checks are made for proper
; inputs. Directives may be given in either lower or upper case.
(define dofmt
(lambda (p cntl args)
  (let ([nmax (- (string-length cntl) 1)])
    (let loop ([n 0] [a args])
      (if (<= n nmax)
        (let ([c (string-ref cntl n)])
          (if (and (char=? c #\~) (< n nmax))
            (case (string-ref cntl (+ n 1))
              [(#\a #\A)
                (display (car a) p)
                (loop (+ n 2) (cdr a))]  
              [(#\s #\S)
                (write (car a) p)
                (loop (+ n 2) (cdr a))]  
              [(#\%)
                (newline p)
                (loop (+ n 2) a)]  
              [(#
                (put-char p #\~) (loop (+ n 2) a)])
            [else
              (put-char p c) (loop (+ n 1) a))]
          (begin
            (put-char p c)
            (loop (+ n 1) a))))))))

; printf and fprintf differ only in that fprintf passes its
; port argument to dofmt while printf passes the current output
; port.
(define printf
  (lambda (control . args)
    (dofmt (current-output-port) control args)))

(define fprintf
  (lambda (p control . args)
    (dofmt p control args)))

**Exercise 12.6.1**

Add error checking to the code for invalid port arguments (fprintf), invalid tilde escapes, and extra or missing arguments.

**Exercise 12.6.2**

Using the optional radix argument to number->string, augment printf and fprintf with support for the following new format directives:

- a. ~b or ~B: print the next unused argument, which must be a number, in binary;
- b. ~o or ~O: print the next unused argument, which must be a number, in octal; and
- c. ~x or ~X: print the next unused argument, which must be a number, in hexadecimal.

For example:

(printf "#x~x #o~o #b~b~%" 16 8 2)

would print
Exercise 12.6.3

Add an "indirect" format directive, \(~@\), that treats the next unused argument, which must be a string, as if it were spliced into the current format string. For example:

```
(printf "---- ~@ ---- " > ~s <" '(a b c))
```

would print

```
---- (a b c) ----
```

Exercise 12.6.4

Implement `format`, a version of `fprintf` that places its output into a string instead of writing to a port. Make use of `object->string` from Exercise 12.5.2 to support the \(~s\) and \(~a\) directives.

```
(let ([x 3] [y 4])
  (format "~s + ~s = ~s" x y (+ x y))) => "3 + 4 = 7"
```

Exercise 12.6.5

Instead of using `object->string`, define `format` using a string output port.

Exercise 12.6.6

Modify `format`, `fprintf`, and `printf` to allow a field size to be specified after the tilde in the \(~a\) and \(~s\) format directives. For example, the directive \(~10s\) would cause the next unused argument to be inserted into the output left-justified in a field of size 10. If the object requires more spaces than the amount specified, allow it to extend beyond the field.

```
(let ([x 'abc] [y '(def)])
  (format "(cons '~5s '~5s) = ~5s" x y (cons x y))) => "(cons 'abc   '(def)) = (abc def)"
```

[Hint: Use `format` recursively.]

Section 12.7. A Meta-Circular Interpreter for Scheme

The program described in this section is a meta-circular interpreter for Scheme, i.e., it is an interpreter for Scheme written in Scheme. The interpreter shows how small Scheme is when the core structure is considered independently from its syntactic extensions and primitives. It also illustrates interpretation techniques that can be applied equally well to languages other than Scheme.

The relative simplicity of the interpreter is somewhat misleading. An interpreter for Scheme written in Scheme can be quite a bit simpler than one written in most other languages. Here are a few reasons why this one is simpler:

- Tail calls are handled properly only because tail calls in the interpreter are handled properly by the host implementation. All that is required is that the interpreter itself be tail-recursive.
- First-class procedures in interpreted code are implemented by first-class procedures in the interpreter, which in turn are supported by the host implementation.
First-class continuations created with \texttt{call/cc} are provided by the host implementation’s \texttt{call/cc}.

Primitive procedures such as \texttt{cons} and \texttt{assq} and services such as storage management are provided by the host implementation.

Converting the interpreter to run in a language other than Scheme may require explicit support for some or all of these items.

The interpreter stores lexical bindings in an \textit{environment}, which is simply an \textit{association list} (see page 165). Evaluation of a \texttt{lambda} expression results in the creation of a procedure within the scope of variables holding the environment and the \texttt{lambda} body. Subsequent application of the procedure combines the new bindings (the actual parameters) with the saved environment.

The interpreter handles only the core syntactic forms described in Section 3.1, and it recognizes bindings for only a handful of primitive procedures. It performs no error checking.

\begin{verbatim}
(interpret 3) \Rightarrow 3

(interpret '(cons 3 4)) \Rightarrow (3 . 4)

(interpret
  '(((lambda (x . y)
      (list x y))
    'a 'b 'c 'd))) \Rightarrow (a (b c d))

(interpret
  '(((call/cc (lambda (k) k))
    (lambda (x) x))
  "HEY!")) \Rightarrow "HEY!"

(interpret
  '(((lambda (memq)
      (memq memq 'a '(b c a d e)))
    (lambda (memq x ls)
      (if (null? ls) #f
        (if (eq? (car ls) x)
          ls
          (memq memq x (cdr ls))))))) \Rightarrow (a d e)

(interpret
  '(((lambda (reverse)
      (set! reverse
        (lambda (ls new)
          (if (null? ls) #f
            (if (eq? (car ls) x)
              ls
              (memq memq x (cdr ls)))))))
    (reverse '(a b c d e) '()))
  #f)) \Rightarrow (e d c b a)

(library (tspl interpreter)
  (export interpret)
  (import (rnrs) (rnrs mutable-pairs))

; primitive-environment contains a small number of primitive
\end{verbatim}
The Scheme Programming Language, 4th Edition

(define primitive-environment
  `(apply . ,apply) (assq . ,assq) (call/cc . ,call/cc)
  (car . ,car) (cadr . ,cadr) (caddr . ,caddr)
  (cadddr . ,cadddr) (cddr . ,cddr) (cdr . ,cdr)
  (cons . ,cons) (eq? . ,eq?) (list . ,list) (map . ,map)
  (memv . ,memv) (null? . ,null?) (pair? . ,pair?)
  (read . ,read) (set-car! . ,set-car!)
  (set-cdr! . ,set-cdr!) (symbol? . ,symbol?)))

; new-env returns a new environment from a formal parameter
; specification, a list of actual parameters, and an outer
; environment.  The symbol? test identifies "improper"
; argument lists.  Environments are association lists,
; associating variables with values.
(define new-env
  (lambda (formals actuals env)
    (cond
      [(null? formals) env]
      [(symbol? formals) (cons (cons formals actuals) env)]
      [else
       (cons
         (cons (car formals) (car actuals))
         (new-env (cdr formals) (cdr actuals) env))])))

; lookup finds the value of the variable var in the environment
; env, using assq.  Assumes var is bound in env.
(define lookup
  (lambda (var env)
    (cdr (assq var env))))

; assign is similar to lookup but alters the binding of the
; variable var by changing the cdr of the association pair
(define assign
  (lambda (var val env)
    (set-cdr! (assq var env) val)))]

; exec evaluates the expression, recognizing a small set of core forms.
(define exec
  (lambda (expr env)
    (cond
      [(symbol? expr) (lookup expr env)]
      [(pair? expr)
       (case (car expr)
         [(quote) (cadr expr)]
         [(lambda)
          (lambda vals
            (let ([env (new-env (cadr expr) vals env)])
              (let loop ([exprs (cddr expr)])
                (if (null? (cdr exprs))
                  (exec (car exprs) env)
                (begin
                  (exec (car exprs) env)
                  (loop (cdr exprs))))))])))

Section 12.7. A Meta-Circular Interpreter for Scheme
The Scheme Programming Language, 4th Edition

[(if)
  (if (exec (cadr expr) env)
    (exec (caddr expr) env)
    (exec (cadddr expr) env))]
[(set!) (assign (cadr expr) (exec (caddr expr) env) env)]
[else
  (apply
    (exec (car expr) env)
    (map (lambda (x) (exec x env)) (cdr expr)))]
[else expr)]

; interpret starts execution with the primitive environment.
(define interpret
  (lambda (expr)
    (exec expr primitive-environment)))

Exercise 12.7.1

As written, the interpreter cannot interpret itself because it does not support several of the syntactic forms used in its implementation: let (named and unnamed), internal define, case, cond, and begin. Rewrite the code for the interpreter, using only the syntactic forms it supports.

Exercise 12.7.2

After completing the preceding exercise, use the interpreter to run a copy of the interpreter, and use the copy to run another copy of the interpreter. Repeat this process to see how many levels deep it will go before the system grinds to a halt.

Exercise 12.7.3

At first glance, it might seem that the lambda case could be written more simply as follows.

[(lambda)
  (lambda vals
    (let ([env (new-env (cadr expr) vals env)])
      (let loop ([exprs (cddr expr)])
        (let ([val (exec (car exprs) env)])
          (if (null? (cdr exprs))
            val
            (loop (cdr exprs)))])))]

Why would this be incorrect? [Hint: What property of Scheme would be violated?]

Exercise 12.7.4

Try to make the interpreter more efficient by looking for ways to ask fewer questions or to allocate less storage space. [Hint: Before evaluation, convert lexical variable references into (access n), where n represents the number of values in the environment association list in front of the value in question.]

Exercise 12.7.5

Scheme evaluates arguments to a procedure before applying the procedure and applies the procedure to the values of these arguments (call-by-value). Modify the interpreter to pass arguments unevaluated and arrange to evaluate them upon reference (call-by-name). [Hint: Use lambda to delay evaluation.] You will need to
create versions of the primitive procedures \((\text{car}, \text{null?}, \text{etc.})\) that take their arguments unevaluated.

**Section 12.8. Defining Abstract Objects**

This example demonstrates a syntactic extension that facilitates the definition of simple abstract objects (see Section 2.9). This facility has unlimited potential as the basis for a complete object-oriented subsystem in Scheme.

Abstract objects are similar to basic data structures such as pairs and vectors. Rather than being manipulated via access and assignment operators, however, abstract objects respond to *messages*. The valid messages and the actions to be taken for each message are defined by code within the object itself rather than by code outside the object, resulting in more modular and potentially more secure programming systems. The data local to an abstract object is accessible only through the actions performed by the object in response to the messages.

A particular type of abstract object is defined with \texttt{define-object}, which has the general form

\begin{verbatim}
(define-object (name var₁ ...) ((var₂ expr) ...) ((msg action) ...))
\end{verbatim}

The first set of bindings \((\text{((var₂ expr) ...)}\) may be omitted. \texttt{define-object} defines a procedure that is called to create new abstract objects of the given type. This procedure is called \texttt{name}, and the arguments to this procedure become the values of the local variables \(\text{var₁ ...}\). After the procedure is invoked, the variables \(\text{var₂ ...}\) are bound to the values \(\text{expr ...}\) in sequence (as with \texttt{let*}) and the messages \(\text{msg ...}\) are bound to the procedure values \(\text{action ...}\) in a mutually recursive fashion (as with \texttt{letrec}). Within these bindings, the new abstract object is created; this object is the value of the creation procedure.

The syntactic form \texttt{send-message} is used to send messages to abstract objects. \(\texttt{(send-message object msg arg ...)}\) sends \texttt{object} the message \texttt{msg} with arguments \texttt{arg ...}. When an object receives a message, the \texttt{arg ...} become the parameters to the action procedure associated with the message, and the value returned by this procedure is returned by \texttt{send-message}.

The following examples should help to clarify how abstract objects are defined and used. The first example is a simple \texttt{kons} object that is similar to Scheme's built-in pair object type, except that to access or assign its fields requires sending it messages.

\begin{verbatim}
(define-object (kons kar kdr)
  ((get-car (lambda () kar))
   (get-cdr (lambda () kdr))
   (set-car! (lambda (x) (set! kar x)))
   (set-cdr! (lambda (x) (set! kdr x)))))
\end{verbatim}

\begin{verbatim}
(define p (kons 'a 'b))
(send-message p get-car) \Rightarrow a
(send-message p get-cdr) \Rightarrow b
(send-message p set-car! 'c)
(send-message p get-cdr) \Rightarrow c
\end{verbatim}

The simple \texttt{kons} object does nothing but return or assign one of the fields as requested. What makes abstract objects interesting is that they can be used to restrict access or perform additional services. The following version of \texttt{kons} requires that a password be given with any request to assign one of the fields. This password is a parameter to the \texttt{kons} procedure.
(define-object (kons kar kdr pwd)
  ((get-car (lambda () kar))
   (get-cdr (lambda () kar))
   (set-car!
     (lambda (x p)
       (if (string=? p pwd)
         (set! kar x))))
   (set-cdr!
     (lambda (x p)
       (if (string=? p pwd)
         (set! kar x)))))

(define p1 (kons 'a 'b "magnificent"))
(send-message p1 set-car! 'c "magnificent")
(send-message p1 get-car) ⇒ c
(send-message p1 set-car! 'd "please")
(send-message p1 get-car) ⇒ c

(define p2 (kons 'x 'y "please"))
(send-message p2 set-car! 'z "please")
(send-message p2 get-car) ⇒ z

One important ability of an abstract object is that it can keep statistics on messages sent to it. The following version of kons counts accesses to the two fields. This version also demonstrates the use of explicitly initialized local bindings.

(define-object (kons kar kdr)
  ((count 0))
  ((get-car
     (lambda ()
       (set! count (+ count 1))
       kar))
   (get-cdr
     (lambda ()
       (set! count (+ count 1))
       kdr))
   (accesses
     (lambda () count))))

(define p (kons 'a 'b))
(send-message p get-car) ⇒ a
(send-message p get-cdr) ⇒ b
(send-message p accesses) ⇒ 2
(send-message p get-cdr) ⇒ b
(send-message p accesses) ⇒ 3

The implementation of define-object is straightforward. The object definition is transformed into a definition of the object creation procedure. This procedure is the value of a lambda expression whose arguments are those specified in the definition. The body of the lambda consists of a let* expression to bind the local variables and a letrec expression to bind the message names to the action procedures. The body of the letrec is another lambda expression whose value represents the new object. The body of this lambda expression compares the messages passed in with the expected messages using a case expression and applies the corresponding action procedure to the remaining arguments.
For example, the definition

\[
\text{(define-object (kons kar kdr)} \\
\quad ((\text{count } 0)) \\
\quad ((\text{get-car} \\
\qquad \text{(lambda ()} \\
\qquad \quad (\text{set! count (+ count 1)}) \\
\qquad \quad \text{kar})) \\
\quad (\text{get-cdr} \\
\qquad \text{(lambda ()} \\
\qquad \quad (\text{set! count (+ count 1)}) \\
\qquad \quad \text{kdr})) \\
\quad (\text{accesses} \\
\qquad \text{(lambda () count))))\]

is transformed into

\[
\text{(define kons} \\
\quad \text{(lambda (kar kdr)} \\
\quad \quad \text{(let* } ([\text{count } 0]) \\
\quad \quad \quad \text{(letrec } ([\text{get-car} \\
\quad \quad \quad \quad \text{(lambda ()} \\
\quad \quad \quad \quad \quad (\text{set! count (+ count 1)}) \text{ kar})] \\
\quad \quad \quad \quad [\text{get-cdr} \\
\quad \quad \quad \quad \text{(lambda ()} \\
\quad \quad \quad \quad \quad (\text{set! count (+ count 1)}) \text{ kdr})] \\
\quad \quad \quad \quad [\text{accesses} \text{(lambda () count)])} \\
\quad \quad \text{(lambda (msg . args)} \\
\quad \quad \quad \quad \text{(case msg} \\
\quad \quad \quad \quad \quad \text{[(get-car) (apply get-car args)]} \\
\quad \quad \quad \quad \quad \text{[(get-cdr) (apply get-cdr args)]} \\
\quad \quad \quad \quad \quad \text{[(accesses) (apply accesses args)]} \\
\quad \quad \quad \quad \quad \text{[else (assertion-violation 'kons} \\
\quad \quad \quad \quad \quad \quad \text{"invalid message"} \\
\quad \quad \quad \quad \quad \quad \text{(cons msg args))])])})\]
\]

\text{(library (tspl oop)} \\
\quad \text{(export define-object send-message)} \\
\quad \text{(import (rnrs))}

; define-object creates an object constructor that uses let* to bind
; local fields and letrec to define the exported procedures.  An
; object is itself a procedure that accepts messages corresponding
; to the names of the exported procedures.  The second pattern is
; used to allow the set of local fields to be omitted.
\text{(define-syntax define-object}
\text{(syntax-rules ()}
\quad \text{[(_ (name . varlist)
\quad \quad \text{((var1 val1) ...)}
\quad \quad \text{((var2 val2) ...))
\quad \quad \text{(define name
\quad \quad \quad \text{(lambda varlist
\quad \quad \quad \quad \text{(let* ([var1 val1] ...) 
\quad \quad \quad \quad \text{(letrec ([var2 val2] ...)}}}

\text{Section 12.8. Defining Abstract Objects 313}
Exercise 12.8.1

Use define-object to define the stack object type from Section 2.9.

Exercise 12.8.2

Use define-object to define a queue object type with operations similar to those described in Section 2.9.

Exercise 12.8.3

It is often useful to describe one object in terms of another. For example, the second kons object type could be described as the same as the first but with a password argument and different actions associated with the set-car! and set-cdr! messages. This is called inheritance; the new type of object is said to inherit attributes from the first. Modify define-object to support inheritance by allowing the optional declaration (inherit object-name) to appear after the message/action pairs. This will require saving some information about each object definition for possible use in subsequent object definitions. Conflicting argument names should be disallowed, but other conflicts should be resolved by using the initialization or action specified in the new object definition.

Exercise 12.8.4

Based on the definition of method on page 317, define a complete object system, but use records rather than vectors to represent object instances. If done well, the resulting object system should be more efficient and easier to use than the system given above.

Section 12.9. Fast Fourier Transform

The procedure described in this section uses Scheme’s complex arithmetic to compute the discrete Fourier transform (DFT) of a sequence of values [4]. Discrete Fourier transforms are used to analyze and process sampled signal sequences in a wide variety of digital electronics applications such as pattern recognition, bandwidth compression, radar target detection, and weather surveillance.

The DFT of a sequence of \( N \) input values,
\{x(n)\}_{m=0}^{N-1},

is the sequence of \(N\) output values,

\{X(m)\}_{m=0}^{N-1},

each defined by the equation

\[ X(m) = \sum_{n=0}^{N-1} x(n)e^{-\frac{2\pi i mn}{N}}. \]

It is convenient to abstract away the constant amount (for given \(N\))

\[ V_N = e^{-\frac{2\pi i N}{N}}, \]

in order to obtain the more concise but equivalent equation

\[ X(m) = \sum_{n=0}^{N-1} x(n)V_N^{mn}. \]

A straightforward computation of the \(N\) output values, each as a sum of \(N\) intermediate values, requires on the order of \(N^2\) operations. A fast Fourier transform (FFT), applicable when \(N\) is a power of 2, requires only on the order of \(N\log_2N\) operations. Although usually presented as a rather complicated iterative algorithm, the fast Fourier transform is most concisely and elegantly expressed as a recursive algorithm.

The recursive algorithm, which is due to Sam Daniel [7], can be derived by manipulating the preceding summation as follows. We first split the summation into two summations and recombine them into one summation from 0 to \(N/2 - 1\).

\[
X(m) = \sum_{n=0}^{N/2-1} x(n)V_N^{mn} + \sum_{n=N/2}^{N-1} x(n)V_N^{mn}
\]

\[
= \sum_{n=0}^{N/2-1} [x(n)V_N^{mn} + x(n+N/2)V_N^{mn(N/2)}]
\]

We then pull out the common factor \(V_N^{mn}\).

\[
X(m) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)V_N^{mn(N/2)}]V_N^{mn}
\]

We can reduce \(V_N^{mn(N/2)}\) to 1 when \(m\) is even and -1 when \(m\) is odd, since

\[
V_N^{nm(N/2)} = \frac{1}{2} - e^{-i\pi m} = \begin{cases} 1, & m \text{ even} \\ -1, & m \text{ odd} \end{cases}
\]

This allows us to specialize the summation for the even and odd cases of \(m = 2k\) and \(m = 2k + 1, 0 \leq k \leq N/2 - 1\).
The resulting summations are DFTs of the $N/2$-element sequences:

$$X(\omega k) = \sum_{n=0}^{N/2-1} [x(n) + x(n + N/2)] W_{N/2}^{kn},$$

$$X(2\omega k + 1) = \sum_{n=0}^{N/2-1} [x(n) - x(n + N/2)] W_{N/2}^{kn}.$$

Thus, the DFT of an $N$-element sequence can be computed recursively by interlacing the DFTs of two $N/2$-element sequences. If we select a base case of two elements, we can describe a recursive fast Fourier transformation (RFFT) algorithm as follows. For $N = 2$,

$$\text{RFFT}[\{x(n)\}]_{n=0}^{N-1} = \{X(\omega k)\}_{k=0}^{N/2-1}$$

$$= \{x(0) + x(1), x(0) - x(1)\} \cdot W_2^0,$$

since $W_2^0 = 1$. For $N > 2$,

$$\text{RFFT}[\{x(n)\}]_{n=0}^{N-1} = \{X(\omega k)\}_{k=0}^{N/2-1}$$

$$= \begin{cases} \text{RFFT}[\{x(n) + x(n + N/2)\}]_{n=0}^{N/2-1}, & m \text{ even} \\ \text{RFFT}[\{x(n) - x(n + N/2)\}]_{n=0}^{N/2-1}, & m \text{ odd} \end{cases}$$

with the attendant interlacing of even and odd components.
The diagram above is adapted from one by Sam Daniel [7] and shows the computational structure of the RFFT algorithm. The first stage computes pairwise sums and differences of the first and second halves of the input; this stage is labeled the \textit{butterfly} stage. The second stage recurs on the resulting subsequences. The third stage interlaces the output of the two recursive calls to RFFT, thus yielding the properly ordered sequence \{X(m)\}_{m=0}^{N-1}.

The procedure \texttt{dft} accepts a sequence (list) of values, \(x\), the length of which is assumed to be a power of 2. \texttt{dft} precomputes a sequence of powers of \(W_N = \frac{1}{\sqrt{N}}e^{-2\pi i/2} = 1\), and calls \texttt{rfft} to initiate the recursion. \texttt{rfft} follows the algorithm outlined above.

\begin{verbatim}
(define (dft x)
  (define (w-powers n)
    (let ([pi (* (acos 0.0) 2)])
      (let ([delta (/ (* -2.0i pi) n)])
        (let f ([n n] [x 0.0])
          (if (= n 0)
            '()
            (cons (exp x) (f (- n 2) (+ x delta)))))))
  (define (evens w)
    (if (null? w)
      '()
      (cons (car w) (evens (cddr w)))))
  (define (interlace x y)
    (if (null? x)
      y
      (interlace (cddr x) (cons (car x) (interlace (cddr y) (cons (cadr x) (cddr y)))))))
  (let ([m (length x)])
    (let ([w (w-powers m)]
          [delta (/ (* -2.0i pi) m)]
          [x (map (lambda (x) (exp (* delta x))) x)])
      (interlace x (rfft x))))
)
\end{verbatim}

The Scheme Programming Language, 4th Edition

Section 12.9. Fast Fourier Transform
'(
  (cons (car x) (cons (car y) (interlace (cdr x) (cdr y))))))
(define (split ls)
  (let split ([fast ls] [slow ls])
    (if (null? fast)
      (values '() slow)
      (let-values ([(front back) (split (cddr fast) (cdr slow))])
        (values (cons (car slow) front) back))))
(define (butterfly x w)
  (let-values ([(front back) (split x)])
    (values
      (map + front back)
      (map * (map - front back) w)))))
(define (rfft x w)
  (if (null? (cddr x))
    (let ([x0 (car x)] [x1 (cadr x)])
      (list (+ x0 x1) (- x0 x1)))
    (let-values ([(front back) (butterfly x w)])
      (let ([w (evens w)])
        (interlace (rfft front w) (rfft back w))))))
(rfft x (w-powers (length x)))

Exercise 12.9.1

Alter the algorithm to employ a base case of four points. What simplifications can be made to avoid multiplying any of the base case outputs by elements of w?

Exercise 12.9.2

Recode dft to accept a vector rather than a list as input, and have it produce a vector as output. Use lists internally if necessary, but do not simply convert the input to a list on entry and the output to a vector on exit.

Exercise 12.9.3

Rather than recomputing the powers of w on each step for a new number of points, the code simply uses the even-numbered elements of the preceding list of powers. Show that doing so yields the proper list of powers. That is, show that (evens (w-powers n)) is equal to (w-powers (/ n 2)).

Exercise 12.9.4

The recursion step creates several intermediate lists that are immediately discarded. Recode the recursion step to avoid any unnecessary allocation.

Exercise 12.9.5

Each element of a sequence of input values may be regenerated from the discrete Fourier transform of the sequence via the equation

\[ x(n) = \frac{1}{N} \sum_{m=0}^{N-1} X(m)e^{2\pi inm/N} \]

Noting the similarity between this equation and the original equation defining \(X(m)\), create a modified version of dft, inverse-dft, that performs the inverse transformation. Verify that (inverse-dft (dft seq)) returns seq for several input sequences seq.
Section 12.10. A Unification Algorithm

Unification \[23\] is a pattern-matching technique used in automated theorem proving, type-inference systems, computer algebra, and logic programming, e.g., Prolog \[6\].

A unification algorithm attempts to make two symbolic expressions equal by computing a unifying substitution for the expressions. A substitution is a function that replaces variables with other expressions. A substitution must treat all occurrences of a variable the same way, e.g., if it replaces one occurrence of the variable \(x\) by \(a\), it must replace all occurrences of \(x\) by \(a\). A unifying substitution, or unifier, for two expressions \(e_1\) and \(e_2\) is a substitution, \(\sigma\), such that \(\sigma(e_1) = \sigma(e_2)\).

For example, the two expressions \(f(x)\) and \(f(y)\) can be unified by substituting \(x\) for \(y\) (or \(y\) for \(x\)). In this case, the unifier \(\sigma\) could be described as the function that replaces \(y\) with \(x\) and leaves other variables unchanged. On the other hand, the two expressions \(x + 1\) and \(y + 2\) cannot be unified. It might appear that substituting \(3\) for \(x\) and \(2\) for \(y\) would make both expressions equal to \(4\) and hence equal to each other. The symbolic expressions, \(3 + 1\) and \(2 + 2\), however, still differ.

Two expressions may have more than one unifier. For example, the expressions \(f(x,y)\) and \(f(1,y)\) can be unified to \(f(1,y)\) with the substitution of \(1\) for \(x\). They may also be unified to \(f(1,5)\) with the substitution of \(1\) for \(x\) and \(5\) for \(y\). The first substitution is preferable, since it does not commit to the unnecessary replacement of \(y\). Unification algorithms typically produce the most general unifier, or mgu, for two expressions. The mgu for two expressions makes no unnecessary substitutions; all other unifiers for the expressions are special cases of the mgu. In the example above, the first substitution is the mgu and the second is a special case.

For the purposes of this program, a symbolic expression can be a variable, a constant, or a function application. Variables are represented by Scheme symbols, e.g., \(x\); a function application is represented by a list with the function name in the first position and its arguments in the remaining positions, e.g., \((f \ x)\); and constants are represented by zero-argument functions, e.g., \((a)\).

The algorithm presented here finds the mgu for two terms, if it exists, using a continuation-passing style, or CPS (see Section 3.4), approach to recursion on subterms. The procedure \texttt{unify} takes two terms and passes them to a help procedure, \texttt{uni}, along with an initial (identity) substitution, a success continuation, and a failure continuation. The success continuation returns the result of applying its argument, a substitution, to one of the terms, i.e., the unified result. The failure continuation simply returns its argument, a message. Because control passes by explicit continuation within \texttt{unify} (always with tail calls), a return from the success or failure continuation is a return from \texttt{unify} itself.

Substitutions are procedures. Whenever a variable is to be replaced by another term, a new substitution is formed from the variable, the term, and the existing substitution. Given a term as an argument, the new substitution replaces occurrences of its saved variable with its saved term in the result of invoking the saved substitution on the argument expression. Intuitively, a substitution is a chain of procedures, one for each variable in the substitution. The chain is terminated by the initial, identity substitution.

\[
\begin{align*}
\text{(unify 'x 'y)} & \Rightarrow y \\
\text{(unify '(f x y) '(g x y))} & \Rightarrow \text{"clash"} \\
\text{(unify '(f x (h)) '(f (h) y))} & \Rightarrow (f (h) (h)) \\
\text{(unify '(f (g x) y) '(f y x))} & \Rightarrow \text{"cycle"} \\
\text{(unify '(f (g x) y) '(f y (g x)))} & \Rightarrow (f (g x) (g x)) \\
\text{(unify '(f (g x) y) '(f y z))} & \Rightarrow (f (g x) (g x)) \\
\end{align*}
\]

\[
\text{(library (tspl unification))} \\
\text{(export unify)} \\
\text{(import (rnrs))}
\]
(define occurs? (lambda (u v)
  (and (pair? v)
    (let f ([l (cdr v)])
      (and (pair? l)
        (or (eq? u (car l))
          (occurs? u (car l))
          (f (cdr l))))))))

(define sigma (lambda (u v s)
  (lambda (x)
    (let f ([x (s x)])
      (if (symbol? x)
        (if (eq? x u) v x)
        (cons (car x) (map f (cdr x)))))))

(define try-subst (lambda (u v s ks kf)
  (let ([u (s u)])
    (if (not (symbol? u))
      (uni u v s ks kf)
      (let ([v (s v)])
        (cond
          [(eq? u v) (ks s)]
          [(occurs? u v) (kf "cycle")]
          [else (ks (sigma u v s))])))))

(define uni (lambda (u v s ks kf)
  (cond
    [(symbol? u) (try-subst u v s ks kf)]
    [(symbol? v) (try-subst v u s ks kf)]
    [(and (eq? (car u) (car v))
      (= (length u) (length v)))
      (let f ([u (cdr u)] [v (cdr v)] [s s])
        (if (null? u)
            (ks s)
            (uni (car u) (car v) s
              (lambda (s) (f (cdr u) (cdr v) s))))]
    ...
    )))
\[(kf)))]
[else (kf "clash")]]

; unify shows one possible interface to uni, where the initial
; substitution is the identity procedure, the initial success
; continuation returns the unified term, and the initial failure
; continuation returns the error message.
(define unify
  (lambda (u v)
    (uni u
     v
     (lambda (x) x)
     (lambda (s) (s u))
     (lambda (msg) msg))))

Exercise 12.10.1

Modify `unify` so that it returns its substitution rather than the unified term. Apply this substitution to both input terms to verify that it returns the same result for each.

Exercise 12.10.2

As mentioned above, substitutions on a term are performed sequentially, requiring one entire pass through the input expression for each substituted variable. Represent the substitution differently so that only one pass through the expression need be made. Make sure that substitutions are performed not only on the input expression but also on any expressions you insert during substitution.

Exercise 12.10.3

Extend the continuation-passing style unification algorithm into an entire continuation-passing style logic programming system.

Section 12.11. Multitasking with Engines

Engines are a high-level process abstraction supporting timed preemption \[10,15\]. Engines may be used to simulate multiprocessing, implement light-weight threads, implement operating system kernels, and perform nondeterministic computations. The engine implementation is one of the more interesting applications of continuations in Scheme.

An engine is created by passing a thunk (procedure of no arguments) to the procedure `make-engine`. The body of the thunk is the computation to be performed by the engine. An engine itself is a procedure of three arguments:

1. \textit{ticks}, a positive integer that specifies the amount of \textit{fuel} to be given to the engine. An engine executes until this fuel runs out or until its computation finishes.

2. \textit{complete}, a procedure of two arguments that specifies what to do if the computation finishes. Its arguments will be the amount of fuel left over and the result of the computation.

3. \textit{expire}, a procedure of one argument that specifies what to do if the fuel runs out before the computation finishes. Its argument will be a new engine capable of continuing the computation from the point of interruption.
When an engine is applied to its arguments, it sets up a timer to fire in \textit{ticks} time units. If the engine computation completes before the timer goes off, the system invokes \texttt{complete}, passing it the number of \textit{ticks} left over and the value of the computation. If, on the other hand, the timer goes off before the engine computation completes, the system creates a new engine from the continuation of the interrupted computation and passes this engine to \texttt{expire}. \texttt{complete} and \texttt{expire} are invoked in the continuation of the engine invocation.

The following example creates an engine from a trivial computation, 3, and gives the engine 10 ticks.

\begin{verbatim}
(define eng
 (make-engine
  (lambda () 3)))
(eng 10
 (lambda (ticks value) value)
 (lambda (x) x)) ⇒ 3
\end{verbatim}

It is often useful to pass \texttt{list} as the \texttt{complete} procedure to an engine, causing the engine to return a list of the ticks remaining and the value if the computation completes.

\begin{verbatim}
(eng 10
 list
 (lambda (x) x)) ⇒ (9 3)
\end{verbatim}

In the example above, the value was 3 and there were 9 ticks left over, i.e., it took only one unit of fuel to evaluate 3. (The fuel amounts given here are for illustration only. The actual amount may differ.)

Typically, the engine computation does not finish in one try. The following example displays the use of an engine to compute the 10th Fibonacci number (see Section 3.2) in steps.

\begin{verbatim}
(define fibonacci
 (lambda (n)
  (if (< n 2)
   n
   (+ (fibonacci (- n 1))
      (fibonacci (- n 2))))))

(define eng
 (make-engine
  (lambda ()
   (fibonacci 10))))

(eng 50
 list
 (lambda (new-eng)
  (set! eng new-eng)
  "expired")) ⇒ "expired"

(eng 50
 list
 (lambda (new-eng)
  (set! eng new-eng)
  "expired")) ⇒ "expired"
\end{verbatim}
Each time the engine's fuel ran out, the expire procedure assigned eng to the new engine. The entire computation required four allotments of 50 ticks to complete; of the last 50 it used all but 23. Thus, the total amount of fuel used was 177 ticks. This leads us to the following procedure, mileage, which uses engines to "time" a computation.

```scheme
(define mileage
  (lambda (thunk)
    (let loop ([eng (make-engine thunk)] [total-ticks 0])
      (eng 50
        (lambda (ticks value)
          (+ total-ticks (- 50 ticks)))
        (lambda (new-eng)
          (loop new-eng (+ total-ticks 50)))))))

(mileage (lambda () (fibonacci 10))) 178
```

The choice of 50 for the number of ticks to use each time is arbitrary, of course. It might make more sense to pass a much larger number, say 10000, in order to reduce the number of times the computation is interrupted.

The next procedure, round-robin, could be the basis for a simple time-sharing operating system. round-robin maintains a queue of processes (a list of engines) and cycles through the queue in a round-robin fashion, allowing each process to run for a set amount of time. round-robin returns a list of the values returned by the engine computations in the order that the computations complete.

```scheme
(define round-robin
  (lambda (engs)
    (if (null? engs)
        '()
        ((car engs) 1
          (lambda (ticks value)
            (cons value (round-robin (cdr engs))))
          (lambda (eng)
            (round-robin
              (append (cdr engs) (list eng))))))))

(round-robin
  (map (lambda (x) (fibonacci 10)))))
```

Assuming the amount of computation corresponding to one tick is constant, the effect of round-robin is to return a list of the values sorted from the quickest to complete to the slowest to complete. Thus, when we call round-robin on a list of engines, each computing one of the Fibonacci numbers, the output list is sorted with the earlier Fibonacci numbers first, regardless of the order of the input list.
More interesting things could happen if the amount of fuel varied each time through the loop. In this case, the computation would be nondeterministic, i.e., the results would vary from call to call.

The following syntactic form, `por` (parallel-or), returns the first of its expressions to complete with a true value. `por` is implemented with the procedure `first-true`, which is similar to round-robin but quits when any of the engines completes with a true value. If all of the engines complete, but none with a true value, `first-true` (and hence `por`) returns `#f`.

```
(define-syntax por
  (syntax-rules ()
    [(_ x ...) (first-true (list (make-engine (lambda () x)) ...))]))
```

```
(define first-true
  (lambda (engs)
    (if (null? engs)
        #f
        ((car engs) 1
         (lambda (ticks value)
          (or value (first-true (cdr engs))))
         (lambda (eng)
          (first-true
           (append (cdr engs) (list eng))))))))
```

Even if one of the expressions is an infinite loop, `por` can still finish (as long as one of the other expressions completes and returns a true value).

```
(por l 2) ⇒ 1
(por ((lambda (x) (x x)) (lambda (x) (x x)))
     (fibonacci 10)) ⇒ 55
```

The first subexpression of the second `por` expression is nonterminating, so the answer is the value of the second subexpression.

Let's turn to the implementation of engines. Any preemptive multitasking primitive must have the ability to interrupt a running process after a given amount of computation. This ability is provided by a primitive timer interrupt mechanism in some Scheme implementations. We will construct a suitable one here.

Our timer system defines three procedures: `start-timer`, `stop-timer`, and `decrement-timer`, which can be operated as follows.

- `(start-timer ticks handler)` sets the timer to `ticks` and installs `handler` as the procedure to be invoked (without arguments) when the timer expires, i.e., reaches zero.
- `(stop-timer)` resets the timer and returns the number of ticks remaining.
- `(decrement-timer)` decrements the timer by one tick if the timer is on, i.e., if it is not zero.
  When the timer reaches zero, `decrement-timer` invokes the saved handler. If the timer has already reached zero, `decrement-timer` returns without changing the timer.
Code to implement these procedures is given along with the engine implementation below.

Using the timer system requires inserting calls to `decrement-timer` in appropriate places. Consuming a timer tick on entry to a procedure usually provides a sufficient level of granularity. This can be accomplished by using `timed-lambda` as defined below in place of `lambda`. `timed-lambda` simply invokes `decrement-timer` before executing the expressions in its body.

```
(define-syntax timed-lambda
  (syntax-rules ()
    [(_ formals exp1 exp2 ...) (lambda formals (decrement-timer) exp1 exp2 ...)]))
```

It may be useful to redefine named `let` and `do` to use `timed-lambda` as well, so that recursions expressed with these constructs are timed. If you use this mechanism, do not forget to use the timed versions of `lambda` and other forms in code run within an engine, or no ticks will be consumed.

Now that we have a suitable timer, we can implement engines in terms of the timer and continuations. We use `call/cc` in two places in the engine implementation: (1) to obtain the continuation of the computation that invokes the engine so that we can return to that continuation when the engine computation completes or the timer expires, and (2) to obtain the continuation of the engine computation when the timer expires so that we can return to this computation if the newly created engine is subsequently run.

The state of the engine system is contained in two variables local to the engine system: `do-complete` and `do-expire`. When an engine is started, the engine assigns to `do-complete` and `do-expire` procedures that, when invoked, return to the continuation of the engine's caller to invoke `complete` or `expire`. The engine starts (or restarts) the computation by invoking the procedure passed as an argument to `make-engine` with the specified number of ticks. The ticks and the local procedure `timer-handler` are then used to start the timer.

Suppose that the timer expires before the engine computation completes. The procedure `timer-handler` is then invoked. It initiates a call to `start-timer` but obtains the ticks by calling `call/cc` with `do-expire`. Consequently, `do-expire` is called with a continuation that, if invoked, will restart the timer and continue the interrupted computation. `do-expire` creates a new engine from this continuation and arranges for the engine's `expire` procedure to be invoked with the new engine in the correct continuation.

If, on the other hand, the engine computation completes before the timer expires, the timer is stopped and the number of ticks remaining is passed along with the value to `do-complete`; `do-complete` arranges for the engine's `complete` procedure to be invoked with the ticks and value in the correct continuation.

Let's discuss a couple of subtle aspects to this code. The first concerns the method used to start the timer when an engine is invoked. The code would apparently be simplified by letting `new-engine` start the timer before it initiates or resumes the engine computation, instead of passing the ticks to the computation and letting it start the timer. Starting the timer within the computation, however, prevents ticks from being consumed prematurely. If the engine system itself consumes fuel, then an engine provided with a small amount of fuel may not progress toward completion. (It may, in fact, make negative progress.) If the software timer described above is used, this problem is actually avoided by compiling the engine-making code with the untimed version of `lambda`.

The second subtlety concerns the procedures created by `do-complete` and `do-expire` and subsequently applied by the continuation of the `call/cc` application. It may appear that `do-complete` could first invoke the engine's `complete` procedure, then pass the result to the continuation (and similarly for `do-expire`) as follows.

```
(escape (complete value ticks))
```
This would result in improper treatment of tail recursion, however. The problem is that the current continuation would not be replaced with the continuation stored in `escape` until the call to the `complete` procedure returns. Consequently, both the continuation of the running engine and the continuation of the engine invocation could be retained for an indefinite period of time, when in fact the actual engine invocation may appear to be tail-recursive. This is especially inappropriate because the engine interface encourages use of continuation-passing style and hence tail recursion. The round-robin scheduler and `first-true` provide good examples of this, since the `expire` procedure in each invokes engines tail-recursively.

We maintain proper treatment of tail recursion by arranging for `do-complete` and `do-expire` to escape from the continuation of the running engine before invoking the `complete` or `expire` procedures. Since the continuation of the engine invocation is a procedure application, passing it a procedure of no arguments results in application of the procedure in the continuation of the engine invocation.

```scheme
define clock 0
(define handler #f)

(define start-timer
  (lambda (ticks new-handler)
    (set! handler new-handler)
    (set! clock ticks)))

(define stop-timer
  (lambda ()
    (let ([time-left clock])
      (set! clock 0)
      time-left)))

(define decrement-timer
  (lambda ()
    (when (> clock 0)
      (set! clock (- clock 1))
      (when (= clock 0) (handler))))))

(define-syntax timed-lambda
  (syntax-rules ()
    [(_ formals exp1 exp2 ...)
      (lambda formals (decrement-timer) exp1 exp2 ...)]))
```

(library (tspl engines)
  (export make-engine timed-lambda)
  (import (rnrs) (tspl timer)))

(define make-engine
  (let ([do-complete #f] [do-expire #f])
    (define timer-handler
      (lambda ()
        (start-timer (call/cc do-expire) timer-handler)))
    (define new-engine
      (lambda (resume)
        (lambda (ticks complete expire)
```

The Scheme Programming Language, 4th Edition

Section 12.11. Multitasking with Engines
The Scheme Programming Language, 4th Edition

((call/cc
 (lambda (escape)
   (set! do-complete
     (lambda (ticks value)
       (escape (lambda () (complete ticks value))))))
 (set! do-expire
   (lambda (resume)
     (escape (lambda ()
       (expire (new-engine resume))))))
 (resume ticks)))

(lambda (proc)
 (new-engine
   (lambda (ticks)
     (start-timer ticks timer-handler)
     (let ([value (proc)])
       (let ([ticks (stop-timer)])
         (do-complete ticks value))))))

(define-syntax timed-lambda
 (syntax-rules ()
  [(_ formals exp1 exp2 ...) (lambda formals (decrement-timer) exp1 exp2 ...)]))

Exercise 12.11.1

If your Scheme implementation allows definition and import of libraries in the interactive top level, try defining the libraries above, then type

(import (rename (tspl engines) (timed-lambda lambda)))

to define make-engine and redefine lambda. Then try out the examples given earlier in this section.

Exercise 12.11.2

It may appear that the nested let expressions in the body of make-engine:

(let ([value (proc)])
  (let ([ticks (stop-timer)])
    (do-complete ticks value)))

could be replaced with the following.

(let ([value (proc)] [ticks (stop-timer)])
  (do-complete value ticks))

Why is this not correct?

Exercise 12.11.3

It would also be incorrect to replace the nested let expressions discussed in the preceding exercise with the following.

(let ([value (proc)])
  (do-complete value (stop-timer)))

Section 12.11. Multitasking with Engines
Why?

Exercise 12.11.4

Modify the engine implementation to provide a procedure, `engine-return`, that returns immediately from an engine.

Exercise 12.11.5

Implement the kernel of a small operating system using engines for processes. Processes should request services (such as reading input from the user) by evaluating an expression of the form `(trap 'request)`. Use `call/cc` and `engine-return` from the preceding exercise to implement `trap`.

Exercise 12.11.6

Write the same operating-system kernel without using engines, building instead from continuations and timer interrupts.

Exercise 12.11.7

This implementation of engines does not allow one engine to call another, i.e., nested engines [10]. Modify the implementation to allow nested engines.
References


References


R. Kent Dybvig / The Scheme Programming Language, Fourth Edition
Copyright © 2009 The MIT Press. Electronically reproduced by permission.
Illustrations © 2009 Jean-Pierre Hébert
ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
Answers to Selected Exercises

Exercise 2.2.1. (page 20)

a. \((+ (* 1.2 (- 2 1/3)) -8.7)\)

b. \((/ (+ 2/3 4/9) (- 5/11 4/3))\)

c. \((+ 1 (/ 1 (+ 2 (/ 1 (+ 1 1/2)))))\)

d. \((* (* (* (* (* (* 1 -2) 3) -4) 5) -6) 7)\) or \((* 1 -2 3 -4 5 -6 7)\)

Exercise 2.2.2. (page 20)
See Section 6.4.

Exercise 2.2.3. (page 20)

a. (car . cdr)

b. (this is silly))

c. (is this silly?)

d. (+ 2 3)

e. (+ 2 3)

f. +

g. (2 3)

h. #<procedure>

i. cons

j. 'cons

k. quote

l. 5

m. 5

n. 5

o. 5

Exercise 2.2.4. (page 21)

(car (cdr (car '((a b) (c d))))) \Rightarrow b
(car (car (cdr '((a b) (c d))))) \Rightarrow c
(car (cdr (car (cdr '((a b) (c d))))) \Rightarrow d

Exercise 2.2.5. (page 21)

'((a . b) ((c) d) ())

Exercise 2.2.6. (page 21)

1 2 3 4 5

1

2

3

4

5
Exercise 2.2.7. (page 21)

(car '((a b) (c d))) \Rightarrow (a b)
(car (car '((a b) (c d)))) \Rightarrow a
(cdr (car '((a b) (c d)))) \Rightarrow (b)
(car (cdr (car '((a b) (c d))))) \Rightarrow b
(cdr (cdr (car '((a b) (c d)))))) \Rightarrow ()
(car (cdr '((a b) (c d)))) \Rightarrow ((c d))
(car (cdr (car '((a b) (c d)))) \Rightarrow (c d)
(car (car (cdr '((a b) (c d)))) \Rightarrow c
(cdr (car (cdr '((a b) (c d)))) \Rightarrow (d)
(car (cdr (car (cdr '((a b) (c d))))) \Rightarrow d
(cdr (cdr (car (cdr '((a b) (c d)))))) \Rightarrow ()
(cdr (cdr '((a b) (c d)))) \Rightarrow ()

Exercise 2.2.8. (page 21)
See Section 2.3.

Exercise 2.3.1. (page 23)

1. Evaluate the variables list, +, −, *, and /, yielding the list, addition, subtraction, multiplication, and division procedures.
2. Apply the list procedure to the addition, subtraction, multiplication, and division procedures, yielding a list containing these procedures in order.
3. Evaluate the variable cdr, yielding the cdr procedure.
4. Apply the cdr procedure to the list produced in step 2, yielding a list containing the subtraction, multiplication, and division procedures.
5. Evaluate the variable car, yielding the car procedure.
6. Apply the car procedure to the list produced in step 4, yielding the subtraction procedure.
7. Evaluate the constants 17 and 5, yielding 17 and 5.
8. Apply the subtraction procedure to 17 and 5, yielding 12.

Other orders are possible. For example, the variable car could have been evaluated before its argument.

Exercise 2.4.1. (page 25)

a. (let ([x (* 3 a)]) (+ (- x b) (+ x b)))
b. (let ([x (list a b c)]) (cons (car x) (cdr x)))

Exercise 2.4.2. (page 25)
The value is 54. The outer let binds x to 9, while the inner let binds x to 3 (9/3). The inner let evaluates to 6 (3 + 3), and the outer let evaluates to 54 (9 × 6).

Exercise 2.4.3. (page 26)

a. (let ([x0 'a] [y0 'b])
   (list (let ([x1 'c]) (cons x1 y0))
   (let ([y1 'd]) (cons x0 y1))))
b. (let ([x0 '((a b) c)])
   (cons (let ([x1 (cdr x0)])
   (car x1))
   (let ([x2 (car x0)])
   (cons (let ([x3 (cdr x2)])
   (car x3))
   (car x0))))
Exercise 2.5.1. (page 30)

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

Exercise 2.5.2. (page 30)
See page 31.

Exercise 2.5.3. (page 30)

a. no free variables
b. +
c. f
d. cons, f, and y
e. cons and y
f. cons, y, and z (y also appears as a bound variable)

Exercise 2.6.1. (page 34)
The program would loop indefinitely.

Exercise 2.6.2. (page 34)

(define compose
  (lambda (p1 p2)
    (lambda (x)
      (p1 (p2 x)))))

(define cadr (compose car cdr))
(define cddr (compose cdr cdr))

Exercise 2.6.3. (page 34)

(define caar (compose car car))
(define cadr (compose car cdr))
(define cdar (compose cdr car))
(define cddr (compose cdr cdr))
(define caaar (compose car caar))
(define caadr (compose car cadr))
(define cadar (compose car cdar))
(define caddr (compose car cddr))
(define cdaar (compose cdr caar))
(define cdadr (compose cdr cadr))
(define cdadr (compose cdr cddr))
(define cdddr (compose cdr cddr))

(define caaaar (compose caar caar))
(define caaadrr (compose caar cadr))
(define caadar (compose caar cdar))
(define caaddr (compose caar cddr))
(define cadaar (compose cadr caar))
(define cadadr (compose cadr cadr))
(define caddar (compose cadr cdar))
(define cadddr (compose cadr cddr))

(define cdaar (compose cdr cdar))
(define cdadr (compose cdr cdr))
(define cdadar (compose cdr cdar))
(define cdaddr (compose cdr cddr))
(define cddaar (compose cddr caar))
(define cddadr (compose cddr cdr))
(define cdddar (compose cddr cdar))
(define cddddr (compose cddr cddr))

Exercise 2.7.1. (page 41)

(define atom?
  (lambda (x)
    (not (pair? x))))

Exercise 2.7.2. (page 41)

(define shorter
  (lambda (ls1 ls2)
    (if (< (length ls2) (length ls1))
      ls2
      ls1)))

Exercise 2.8.1. (page 46)
The structure of the output would be the mirror image of the structure of the input. For example, (a . b) would become (b . a) and ((a . b) . (c . d)) would become ((d . c) . (b . a)).

Exercise 2.8.2. (page 46)

(define append
  (lambda (ls1 ls2)
    (if (null? ls1)
      ls2
      (cons (car ls1) (append (cdr ls1) ls2))))))

Exercise 2.8.3. (page 46)

(define make-list
  (lambda (n x)
    (if (= n 0)
      '
      (cons x (make-list (- n 1) x)))))
Exercise 2.8.4. (page 47)
See the description of list-ref on page 160 and the description of list-tail on page 160.

Exercise 2.8.5. (page 47)

(define shorter?
  (lambda (ls1 ls2)
    (and (not (null? ls2))
      (or (null? ls1)
        (shorter? (cdr ls1) (cdr ls2))))))

(define shorter
  (lambda (ls1 ls2)
    (if (shorter? ls2 ls1)
        ls2
        ls1)))

Exercise 2.8.6. (page 47)

(define even?
  (lambda (x)
    (or (= x 0)
      (odd? (- x 1)))))

(define odd?
  (lambda (x)
    (and (not (= x 0))
      (even? (- x 1)))))

Exercise 2.8.7. (page 47)

(define transpose
  (lambda (ls)
    (cons (map car ls) (map cdr ls))))

Exercise 2.9.1. (page 54)

(define make-counter
  (lambda (init incr)
    (let ([next init])
      (lambda ()
        (let ([v next])
          (set! next (+ next incr))
          v)))))

Exercise 2.9.2. (page 55)

(define make-stack
  (lambda ()
    (let ([ls '()]
      (lambda (msg . args)
        (case msg
          [(empty? mt?) (null? ls)]
          [(push!) (set! ls (cons (car args) ls))]
          [(top) (car ls)]))
      )
    )
  )
Exercise 2.9.3. (page 55)

(define make-stack
  (lambda ()
    (let ([ls '()])
      (lambda (msg . args)
        (case msg
          [(empty? mt?) (null? ls)]
          [(push!) (set! ls (cons (car args) ls))]
          [(top) (car ls)]
          [(pop!) (set! ls (cdr ls))]
          [(ref) (list-ref ls (car args))]
          [(set!) (set-car! (list-tail ls (car args)) (cadr args))]
          [else "oops"])))
))

Exercise 2.9.4. (page 55)

(define make-stack
  (lambda (n)
    (let ([v (make-vector n)] [i -1])
      (lambda (msg . args)
        (case msg
          [(empty? mt?) (= i -1)]
          [(push!) (set! i (+ i 1))
            (vector-set! v i (car args))]
          [(top) (vector-ref v i)]
          [(pop!) (set! i (- i 1))]
          [(ref) (vector-ref v (- i (car args)))]
          [(set!) (vector-set! v (- i (car args)) (cadr args))]
          [else "oops"])))
))

Exercise 2.9.5. (page 56)

(define emptyq?
  (lambda (q)
    (eq? (car q) (cdr q))))

(define getq
  (lambda (q)
    (if (emptyq? q)
      (assertion-violation 'getq "the queue is empty")
      (car (car q))))
)

(define delq!
  (lambda (q)
    (if (emptyq? q)
      (assertion-violation 'delq! "the queue is empty")
      (set-car! q (cdr (car q))))))

Exercise 2.9.6. (page 56)
(define make-queue
  (lambda ()
    (cons '() '())))

(define putq!
  (lambda (q v)
    (let ([p (cons v '())])
      (if (null? (car q))
          (begin
            (set-car! q p)
            (set-cdr! q p))
          (begin
            (set-cdr! (cdr q) p)
            (set-cdr! q p))))))

(define getq
  (lambda (q)
    (car (car q))))

(define delq!
  (lambda (q)
    (if (eq? (car q) (cdr q))
        (begin
          (set-car! q '())
          (set-cdr! q '())
          (set-car! q (cdr (car q))))))

Exercise 2.9.7. (page 56)
When asked to print a cyclic structure, some implementations print a representation of the output that reflects
its cyclic structure. Other implementations do not detect the cycle and produce either no output or an infinite
stream of output. When length is passed a cyclic list, an exception is raised, likely with a message
indicating that the list is not proper. The definition of length on page 42 will, however, simply loop
indefinitely.

Exercise 2.9.8. (page 56)

(define race
  (lambda (hare tortoise)
    (if (pair? hare)
        (let ([hare (cdr hare)])
          (if (pair? hare)
              (and (not (eq? hare tortoise))
                (race (cdr hare) (cdr tortoise)))
              (null? hare))
            (null? hare))))

(define list?
  (lambda (x)
    (race x x)))

Exercise 3.1.1. (page 64)

(let ([x (memv 'a ls)]) (and x (memv 'b x))) →
  ((lambda (x) (and x (memv 'b x))) (memv 'a ls)) →
Exercise 3.1.2. (page 64)

\[
(\lambda (x) (\text{if } x \ (\text{memv 'b x})) \ #f)) \ (\text{memv 'a ls}) \\
(\lambda (x) (\text{if } x \ (\text{memv 'b x})) \ #f)) \ (\text{memv 'a ls})
\]

Exercise 3.1.3. (page 64)

See page 97.

Exercise 3.1.4. (page 64)

\[
(\text{define-syntex when} \\
(\text{syntax-rules ()} \\
[(_ e0 e1 e2 ...) \ (if e0 (begin e1 e2 ...))]))
\]

\[
(\text{define-syntex unless} \\
(\text{syntax-rules ()} \\
[(_ e0 e1 e2 ...) \ (when (not e0) e1 e2 ...)]))
\]

Exercise 3.2.1. (page 72)


Exercise 3.2.2. (page 72)

\[
(\text{define factor} \\
(\lambda (n) \\
(\text{letrec (}} [ f (\lambda (n i)) \\
(\text{cond} \\
[ (\geq i n) \ (\text{list n})] \\
[ (\text{integer?} (/ n i)) \\
 (\text{cons i (f (/ n i) i))]) \\
[ \text{else (f n (+ i 1))}])]) \\
(f n 2)))))
\]

Exercise 3.2.3. (page 72)

Yes, but we need two named let expressions, one for even? and one for odd?.

\[
(\text{let even? ([x 20])} \\
(\text{or (= x 0)} \\
 (\text{let odd? ([x (- x 1)])} \\
 (\text{and (not (= x 0))} \\
 (\text{even? (- x 1))})))
\]

Exercise 3.2.4. (page 72)

\[
(\text{define fibcount1 0} \\
(\text{define fibonacci1})
\]

Answers to Selected Exercises
(lambda (n)
  (let fib ([i n])
    (set! fibcount1 (+ fibcount1 1))
    (cond
     [(= i 0) 0]
     [(= i 1) 1]
     [else (+ (fib (- i 1)) (fib (- i 2)))])))))

(define fibcount2 0)
(define fibonacci2
  (lambda (n)
    (if (= n 0)
      0
      (let fib ([i n] [a1 1] [a2 0])
        (set! fibcount2 (+ fibcount2 1))
        (if (= i 1)
          a1
          (fib (- i 1) (+ a1 a2) a1))))))

The counts for (fibonacci 10) are 177 and 10, for (fibonacci 20) are 21891 and 20, and for (fibonacci 30) are 2692537 and 30. While the number of calls made by the second is directly proportional to the input, the number of calls made by the first grows rapidly (exponentially, in fact) as the input value increases.

Exercise 3.2.5. (page 73)
See page 312.

Exercise 3.2.6. (page 73)
A call in the last subexpression of an or expression in tail position would not be a tail call with the modified definition of or. For the even?/odd? example, the resulting definition of even? would no longer be tail-recursive and for very large inputs might exhaust available space.

The expansion performed by this definition is incorrect in another way, which has to do with multiple return values (Section 5.8): if the last subexpression returns multiple values, the or expression should return multiple values, but with the incorrect definition, each subexpression appears on the right-hand side of a let, which expects a single return value. The simpler and incorrect definition of and has the same problem.

Exercise 3.2.7. (page 73)
The first of the three versions of factor below directly addresses the identified problems by stopping at \( \sqrt{n} \), avoiding the redundant division, and skipping the even factors after 2. Stopping at \( \sqrt{n} \) probably yields the biggest savings, followed by skipping even factors greater than 2. Avoiding the redundant division is less important, since it occurs only when a factor is found.

(define factor
  (lambda (n)
    (let f ([n n] [i 2] [step 1])
      (if (> i (sqrt n))
        (list n)
        (let ([n/i (/ n i)])
          (if (integer? n/i)
            (cons i (f n/i i step))
            (f n (+ i step) 2)))))))

The second version replaces (> i (sqrt n)) with (> (* i i) n), since * is typically much faster
than sqrt.

(define factor
  (lambda (n)
    (let f ([n n] [i 2] [step 1])
      (if (> (* i i) n)
        (list n)
        (let ([n/i (/ n i)])
          (if (integer? n/i)
            (cons i (f n/i i step))
            (f n (+ i step) 2)))))))

The third version uses gcd (see page 179) to avoid most of the divisions, since gcd should be faster than /.

(define factor
  (lambda (n)
    (let f ([n n] [i 2] [step 1])
      (if (> (* i i) n)
        (list n)
        (if (= (gcd n i) 1)
          (f n (+ i step) 2)
          (cons i (f (/ n i) i step)))))))

To see the difference these changes make, time each version of factor, including the original, in your Scheme system to see which performs better. Try a variety of inputs, including larger ones like (+ (expt 2 100) 1).

Exercise 3.3.1. (page 77)

(let ([k.n (call/cc (lambda (k) (cons k 0)))])
  (let ([k (car k.n)] [n (cdr k.n)])
    (write n)
    (newline)
    (k (cons k (+ n 1)))))

Or with multiple values (see Section 5.8):

(call-with-values
  (lambda () (call/cc (lambda (k) (values k 0))))
  (lambda (k n)
    (write n)
    (newline)
    (k k (+ n 1)))))

Exercise 3.3.2. (page 77)

(define product
  (lambda (ls)
    (if (null? ls)
      1
      (if (= (car ls) 0)
        0
        (let ([n (product (cdr ls))])
          (if (= n 0) 0 (* n (car ls))))))))
Exercise 3.3.3. (page 77)
If one of the processes returns without calling `pause`, it returns to the call to `pause` that first caused it to run, or to the original call to `start` if it was the first process in the list. Here is a reimplementation of the system that allows a process to `quit` explicitly. If other processes are active, the `lwp` system continues to run. Otherwise, control returns to the continuation of the original call to `start`.

```
(define lwp-list '())
(define lwp
  (lambda (thunk)
    (set! lwp-list (append lwp-list (list thunk))))))
(define start
  (lambda ()
    (call/cc
      (lambda (k)
        (set! quit-k k)
        (next)))))
(define next
  (lambda ()
    (let ([p (car lwp-list)])
      (set! lwp-list (cdr lwp-list))
      (p)))))
(define pause
  (lambda ()
    (call/cc
      (lambda (k)
        (lwp (lambda () (k #f)))
        (next)))))
(define quit
  (lambda (v)
    (if (null? lwp-list)
      (quit-k v)
      (next)))))

Exercise 3.3.4. (page 77)

```

```
(define lwp-queue (make-queue))
(define lwp
  (lambda (thunk)
    (putq! lwp-queue thunk)))
(define start
  (lambda ()
    (let ([p (getq lwp-queue)])
      (delq! lwp-queue)
      (p)))))
(define pause
  (lambda ()
    (call/cc
      (lambda (k)
        (lwp (lambda () (k #f)))
        (start))))))

Exercise 3.4.1. (page 80)

```

```
(define reciprocal

Answers to Selected Exercises
(lambda (n success failure)
  (if (= n 0)
      (failure)
      (success (/ 1 n))))

Exercise 3.4.2. (page 80)

(define retry #f)
(define factorial
  (lambda (x)
    (let f ([x x] [k (lambda (x) x)])
      (if (= x 0)
          (begin (set! retry k) (k 1))
          (f (- x 1) (lambda (y) (k (* x y)))))))

Exercise 3.4.3. (page 80)

(define map/k
  (lambda (p ls k)
    (if (null? ls)
        (k '())
        (p (car ls)
            (lambda (x)
              (lambda (x)
                (map/k p (cdr ls)
                  (lambda (ls)
                    (k (cons x ls))))))))

(define reciprocals
  (lambda (ls)
    (map/k (lambda (x k) (if (= x 0) "zero found" (k (/ 1 x))))
      ls
      (lambda (x) x))))

Exercise 3.5.1. (page 85)

(define-syntax complain
  (syntax-rules ()
    [(_ ek msg expr) (ek (list msg expr))]))

Exercise 3.5.2. (page 85)

(define calc
  (lambda (expr)
    (call/cc
      (lambda (ek)
        (define do-calc
          (lambda (expr)
            (cond
              [(number? expr) expr]
              [(and (list? expr) (= (length expr) 3))
                (let ([op (car expr)] [args (cdr expr)])
                  (case op
                    [(add) (apply-op + args)]
...
Exercise 3.5.3. (page 85)

(define calc #f)
(let ()
  (define do-calc
    (lambda (expr)
      (cond
        [(number? expr) expr]
        [(and (list? expr) (= (length expr) 3))
          (let ([op (car expr)] [args (cdr expr)])
            (case op
              [(add) (apply-op + args)]
              [(sub) (apply-op - args)]
              [(mul) (apply-op * args)]
              [(div) (apply-op / args)]
              [else (complain "invalid operator" op)])]
          [else (complain "invalid expression" expr)])
      (define apply-op
        (lambda (op args)
          (op (do-calc (car args)) (do-calc (cadr args))))))
      (define complain
        (lambda (msg expr)
          (assertion-violation 'calc msg expr)))
      (set! calc
        (lambda (expr)
          (do-calc expr))))))}

Exercise 3.5.4. (page 85)
This adds sqrt, times (an alias for mul), and expt along with minus.

(let ()
  (define do-calc
    (lambda (ek expr)
      (cond
        [(number? expr) expr]
        [(and (list? expr) (= (length expr) 2))
          (let ([op (car expr)] [args (cdr expr)])
            (case op
              [(minus) (apply-op1 ek - args)]
              [(sqrt) (apply-op1 ek sqrt args)]
              [else (complain ek "invalid unary operator" op)])])))

Answers to Selected Exercises
Exercise 3.6.1. (page 87)
This version of gpa returns x when all of the input letter grades are x.

(define-syntax gpa
  (syntax-rules ()
    [(_ g1 g2 ...)\[
      (let ([ls (map letter->number (remq 'x '(g1 g2 ...)))])
        (if (null? ls)
            'x
            (/ (apply + ls) (length ls))))]
    ))

Exercise 3.6.2. (page 87)
After defining $distribution and distribution within the library as follows:

(define $distribution
  (lambda (ls)
    (let loop ([ls ls] [a 0] [b 0] [c 0] [d 0] [f 0])
      (if (null? ls)
          (list (list a 'a) (list b 'b) (list c 'c)
                (list d 'd) (list f 'f))
        (case (car ls)
          [(a) (loop (cdr ls) (+ a 1) b c d f)]
          [(b) (loop (cdr ls) a (+ b 1) c d f)]
          [(c) (loop (cdr ls) a b (+ c 1) d f)]
          [(d) (loop (cdr ls) a b c (+ d 1) f)]
          [(f) (loop (cdr ls) a b c d (+ f 1))]
          ; ignore x grades, per preceding exercise
          [(x) (loop (cdr ls) a b c d f)]
          [else (assertion-violation 'distribution)]))))

Answers to Selected Exercises
modify the export line to add distribution (but not $distribution).

Exercise 3.6.3. (page 87)
After defining histogram as follows:

(define histogram
  (lambda (port distr)
    (for-each
      (lambda (n g)
        (put-datum port g)
        (put-string port ": ")
        (let loop ([n n])
          (unless (= n 0)
            (put-char port #\*)
            (loop (- n 1)))))
      (map car distr)
      (map cadr distr))))

modify the export line to add histogram. The solution uses for-each, which is described on page 118.
Formal Syntax

The formal grammars and accompanying text appearing here describe the written syntax of Scheme data values, or datums. The grammars also effectively cover the written syntax of Scheme syntactic forms, since every Scheme syntactic form has a representation as a Scheme datum. In particular, parenthesized syntactic forms are written as lists, and identifiers (e.g., keywords and variables) are written as symbols. The high-level structure of each syntactic form is described in detail by the entries marked "syntax" in Chapters 4 through 11, and the syntactic forms are summarized in the Summary of Forms.

The written representation of a datum involves tokens, whitespace, and comments. Tokens are sequences of one or more characters representing atomic datums or serving as punctuation marks. The tokens that represent atomic datums are symbols, numbers, strings, booleans, and characters, while the tokens serving as punctuation marks are open and close parentheses, open and close brackets, the open vector parenthesis # (, the open bytevector parenthesis #vu8 (, the dotted pair marker . (dot), the quotation marks ', and `, the unquotation marks , and @, the syntax quotation marks # ' and # `, and the syntax unquotation marks #, and #, @.

Whitespace consists of space, tab, newline, form-feed, carriage-return, and next-line characters along with any additional characters categorized as Zs, Zl, or Zp by the Unicode standard [30]. A newline character is also called a linefeed character. Some whitespace characters or character sequences serve as line endings, which are recognized as part of the syntax of line comments and strings. A line ending is a newline character, a next-line character, a line-separator character, a carriage-return character followed by a newline character, a carriage return followed by a next-line character, or a carriage return not followed by a newline or next-line character. A different set of whitespace characters serve as intraline whitespace, which are recognized as part of the syntax of strings. Intraline whitespace includes spaces, tabs, and any additional Unicode characters whose general category is Zs. The sets of intraline whitespace characters and line endings are disjoint, and there are other whitespace characters, such as form feed, that are not in either set.

Comments come in three flavors: line comments, block comments, and datum comments. A line comment consists of a semicolon (; ) followed by any number of characters up to the next line ending or end of input. A block comment consists of a #| prefix, any number of characters and nested block comments, and a |# suffix. A datum comment consists of a #; prefix followed by any datum.

Symbols, numbers, characters, booleans, and the dotted pair marker ( . ) must be delimited by the end the input, whitespace, the start of a comment, an open or close parenthesis, an open or close bracket, a string quote ( " ), or a hash mark ( # ). Any token may be preceded or followed by any number of whitespace characters and comments.

Case is significant in the syntax of characters, strings, and symbols except within a hex scalar value, where the hexadecimal digits "a" through "f" may be written in either upper or lower case. (Hex scalar values are hexadecimal numbers denoting Unicode scalar values.) Case is insignificant in the syntax of booleans and numbers. For example, Hello is distinct from hello, \A is distinct from \a, and "String" is distinct from "string", while #T is equivalent to #t, #E1E3 is equivalent to #e1e3, #X2abc is equivalent to \x2abc, and \x3BA is equivalent to \x3ba.

A conforming implementation of the Revised Report is not permitted to extend the syntax of datums, with one exception: it is permitted to recognize any token starting with the prefix # ! as a flag indicating certain extensions are valid in the text following the flag. So, for example, an implementation might recognize the flag #!braces and switch to a mode in which lists may be enclosed in braces as well as in parentheses and brackets.

#!braces ' { a b c } ⇒ ( a b c )
The flag `#!r6rs` may be used to declare that the subsequent text is written in R6RS syntax. It is good practice to include `#!r6rs` at the start of any file containing a portable library or top-level program to specify that R6RS syntax is being used, in the event that future reports extend the syntax in ways that are incompatible with the text of the library or program. `#!r6rs` is otherwise treated as a comment.

In the grammars appearing below, `<empty>` stands for an empty sequence of characters. An item followed by an asterisk (\*\) represents zero or more occurrences of the item, and an item followed by a raised plus sign (\+\) represents one or more occurrences. Spacing between items within a production appears for readability only and should be treated as if it were not present.

**Datums.** A datum is a boolean, character, symbol, string, number, list, vector, or bytevector.

\[
<datum> \rightarrow <boolean> \\
    | <character> \\
    | <symbol> \\
    | <string> \\
    | <number> \\
    | <list> \\
    | <vector> \\
    | <bytevector>
\]

Lists, vectors, and bytevectors are compound datums formed from groups of tokens possibly separated by whitespace and comments. The others are single tokens.

**Booleans.** Boolean false is written `#f`. While all other values count as true, the canonical true value (and only other value to be considered a boolean value by the `boolean?` predicate) is written `#t`.

\[
<boolean> \rightarrow #t | #f
\]

Case is not significant in the syntax of booleans, so these may also be written as `#T` and `#F`.

**Characters.** A character object is written as the prefix `#\` followed by a single character, a character name, or a sequence of characters specifying a Unicode scalar value.

\[
<character> \rightarrow #\ <any character> | #\ <character name> | #x <hex scalar value>
\]

A `#\` prefix followed by a character name is always interpreted as a named character, e.g., `#\page` is treated as `#\page` rather than `#\p` followed by the symbol `age`. Characters must also be delimited, as described above, so that `#\pager` is treated as a syntax error rather than as the character `#\p` followed by the symbol `ager` or the character `#\page` followed by the symbol `r`.

Case is significant in the syntax of character objects, except within a hex scalar value.

**Strings.** A string is written as a sequence of string elements enclosed in string quotes (double quotes). Any character other than a string quote or backslash can appear as a string element. A string element can also
consist of a backslash followed by a single character, a backslash followed by sequence of characters specifying a Unicode scalar value, or a backslash followed by sequence of intraline whitespace characters that includes a single line ending.

\[<\text{string}> \rightarrow " <\text{string character}>* " \]
\[<\text{string element}> \rightarrow <\text{any character except } " \text{ or } \backslash> \]
\[\ | \" | <\text{x} <\text{hex scalar value}> ; | \] <intraline whitespace>* <line ending> <intraline whitespace>*\]

A string element consisting of a single character represents that character, except that any single character or pair of characters representing a line ending represents a single newline character. A backslash followed by a double quote represents a double quote, while a backslash followed by a backslash represents a backslash. A backslash followed by a represents the alarm character (U+0007); by b, backspace (U+0008); by f, form feed (U+000C); by n, newline (U+000A); by r, carriage return (U+000D); by t, tab (U+0009); and by v, vertical tab (U+000B). A backslash followed by x, a hex scalar value, and a semi-colon (;) represents the Unicode character specified by the scalar value. The <hex scalar value> nonterminal is defined under Characters above. Finally, a sequence of characters consisting of a backslash followed by intraline whitespace that includes a single line ending represents no characters.

Case is significant in the syntax of strings, except within a hex scalar value.

**Symbols.** A symbol is written either as an "initial" character followed by a sequence of "subsequent" characters or as a "peculiar symbol." Initial characters are letters, certain special characters, an additional set of Unicode characters, or arbitrary characters specified by Unicode scalar values. Subsequent characters are initial characters, digits, certain additional special characters, and a set of additional Unicode characters. The peculiar symbols are +, -, . . . , and any sequence of subsequent characters prefixed by ->.

\[<\text{symbol}> \rightarrow <\text{initial}> <\text{subsequent}>* \]
\[<\text{initial}> \rightarrow <\text{letter}> | ! | $ | % | & | * | / | : | ; | = | ? | \text{^} | _ | <\text{Unicode Lu, Ll, Lt, Lm, Lo, Mn, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co}> \]
\[\ | <\text{x} <\text{hex scalar value}> ; | \]
\[<\text{subsequent}> \rightarrow <\text{initial}> | <\text{digit 10}> | . | . | @ | <\text{Unicode Nd, Mc, or Me}> \]
\[<\text{letter}> \rightarrow a | b | ... | z | A | B | ... | Z \]

<Unicode Lu, Ll, Lt, Lm, Lo, Mn, Ni, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co> represents any character whose Unicode scalar value is greater than 127 and whose Unicode category is one of the listed categories. <Unicode Nd, Mc, or Me> represents any character whose Unicode category is one of the listed categories. The <hex scalar value> nonterminal is defined under Characters above, and <digit 10> is defined under Numbers below.

Case is significant in symbols.

**Numbers.** Numbers can appear in one of four radices: 2, 8, 10, and 16, with 10 the default. Several of the productions below are parameterized by the radix, r, and each represents four productions, one for each of the four possible radices. Numbers that contain radix points or exponents are constrained to appear in radix 10, so <decimal r> is valid only when r is 10.

\[<\text{number}> \rightarrow <\text{num 2}> | <\text{num 8}> | <\text{num 10}> | <\text{num 16}> \]
\[<\text{num r}> \rightarrow <\text{prefix r}> <\text{complex r}> \]
\[<\text{prefix r}> \rightarrow <\text{radix r}> <\text{exactness}> | <\text{exactness}> <\text{radix r}> \]
\[<\text{radix 2}> \rightarrow \#b \]
\[<\text{radix 8}> \rightarrow \#o \]
A number written as above is inexact if it is prefixed by #i or if it is not prefixed by #e and contains a decimal point, nonempty exponent, or nonempty mantissa width. Otherwise, it is exact.

Case is not significant in the syntax of numbers.

Lists. Lists are compound datums formed from groups of tokens and possibly involving other datums, including other lists. Lists are written as a sequence of datums within parentheses or brackets; as a nonempty sequence of datums, dotted-pair marker ( . ), and single datum enclosed within parentheses or brackets; or as an abbreviation.

If no dotted-pair marker appears in a list enclosed in parentheses or brackets, it is a proper list, and the datums are the elements of the list, in the order given. If a dotted-pair marker appears, the initial elements of the list are those before the marker, and the datum that follows the marker is the tail of the list. The dotted-pair marker is typically used only when the datum that follows the marker is not itself a list. While any proper list may be written without a dotted-pair marker, a proper list can be written in dotted-pair notation by placing a list after the dotted-pair marker.

The abbreviations are equivalent to the corresponding two-element lists shown below. Once an abbreviation has been read, the result is indistinguishable from its nonabbreviated form.

' <datum> → (quote <datum>)
' <datum> → (quasiquote <datum>)
Vectors. Vectors are compound datums formed from groups of tokens and possibly involving other datums, including other vectors. A vector is written as an open vector parenthesis followed by a sequence of datums and a close parenthesis.

\[
\text{<vector>} \rightarrow \#(\text{<datum>*})
\]

Bytevectors. Bytevectors are compound datums formed from groups of tokens, but the syntax does not permit them to contain arbitrary nested datums. A bytevector is written as an open bytevector parenthesis followed by a sequence of octets (unsigned 8-bit exact integers) and a close parenthesis.

\[
\text{<bytevector>} \rightarrow \#\text{vu8(<octet>*})
\]

\[
\text{<octet>} \rightarrow \text{<any <number> representing an exact integer } n, 0 \leq n \leq 255>
\]
# Summary of Forms

The table that follows summarizes the Scheme syntactic forms and procedures described in Chapters 4 through 11. It shows the category of the form and the page number where it is defined. The category states whether the form describes a syntactic form or a procedure.

All page numbers appearing here refer to the printed version of this book and also serve as hypertext links to the corresponding locations in the electronic version of this book.

<table>
<thead>
<tr>
<th>Form</th>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>`obj</td>
<td>syntax</td>
<td>141</td>
</tr>
<tr>
<td>`obj</td>
<td>syntax</td>
<td>142</td>
</tr>
<tr>
<td>,obj</td>
<td>syntax</td>
<td>142</td>
</tr>
<tr>
<td>,@obj</td>
<td>syntax</td>
<td>142</td>
</tr>
<tr>
<td>=&gt;</td>
<td>syntax</td>
<td>112</td>
</tr>
<tr>
<td>_</td>
<td>syntax</td>
<td>297</td>
</tr>
<tr>
<td></td>
<td>syntax</td>
<td>297</td>
</tr>
<tr>
<td>#'template</td>
<td>syntax</td>
<td>300</td>
</tr>
<tr>
<td>#`template</td>
<td>syntax</td>
<td>305</td>
</tr>
<tr>
<td>#,template</td>
<td>syntax</td>
<td>305</td>
</tr>
<tr>
<td>#,@template</td>
<td>syntax</td>
<td>305</td>
</tr>
<tr>
<td>&amp;assertion</td>
<td>syntax</td>
<td>366</td>
</tr>
<tr>
<td>&amp;condition</td>
<td>syntax</td>
<td>362</td>
</tr>
<tr>
<td>&amp;error</td>
<td>syntax</td>
<td>367</td>
</tr>
<tr>
<td>&amp;i/o</td>
<td>syntax</td>
<td>371</td>
</tr>
<tr>
<td>&amp;i/o-decoding</td>
<td>syntax</td>
<td>375</td>
</tr>
<tr>
<td>&amp;i/o-encoding</td>
<td>syntax</td>
<td>376</td>
</tr>
<tr>
<td>&amp;i/o-file-already-exists</td>
<td>syntax</td>
<td>374</td>
</tr>
<tr>
<td>&amp;i/o-file-does-not-exist</td>
<td>syntax</td>
<td>374</td>
</tr>
<tr>
<td>&amp;i/o-file-is-read-only</td>
<td>syntax</td>
<td>374</td>
</tr>
<tr>
<td>&amp;i/o-file-protection</td>
<td>syntax</td>
<td>373</td>
</tr>
<tr>
<td>&amp;i/o-filename</td>
<td>syntax</td>
<td>373</td>
</tr>
<tr>
<td>&amp;i/o-invalid-position</td>
<td>syntax</td>
<td>372</td>
</tr>
<tr>
<td>&amp;i/o-port</td>
<td>syntax</td>
<td>375</td>
</tr>
<tr>
<td>&amp;i/o-read</td>
<td>syntax</td>
<td>372</td>
</tr>
<tr>
<td>&amp;i/o-write</td>
<td>syntax</td>
<td>372</td>
</tr>
<tr>
<td>&amp;implementation-restriction</td>
<td>syntax</td>
<td>369</td>
</tr>
<tr>
<td>&amp;irritants</td>
<td>syntax</td>
<td>368</td>
</tr>
<tr>
<td>&amp;lexical</td>
<td>syntax</td>
<td>370</td>
</tr>
<tr>
<td>&amp;message</td>
<td>syntax</td>
<td>368</td>
</tr>
<tr>
<td>&amp;no-infinities</td>
<td>syntax</td>
<td>376</td>
</tr>
<tr>
<td>&amp;no-nans</td>
<td>syntax</td>
<td>377</td>
</tr>
<tr>
<td>&amp;non-continuable</td>
<td>syntax</td>
<td>369</td>
</tr>
<tr>
<td>&amp;serious</td>
<td>syntax</td>
<td>366</td>
</tr>
<tr>
<td>&amp;syntax</td>
<td>syntax</td>
<td>370</td>
</tr>
</tbody>
</table>

Summary of Forms 355
&undefined
&violation
&warning
&who
(* num ...)
(+ num ...)
(- num)
(- num₁ num₂ num₃ ...)
(/ num)
(/ num₁ num₂ num₃ ...)
(< real₁ real₂ real₃ ...)
(<= real₁ real₂ real₃ ...)
(= num₁ num₂ num₃ ...)
(> real₁ real₂ real₃ ...)
(>= real₁ real₂ real₃ ...)
(abs real)
(acos num)
(and expr ...)
(angle expr)
(append)
(append list ... obj)
(apply procedure obj ... list)
(asin num)
(assert expression)
(assertion-violation who msg irritant ...)
(assertion-violation? obj)
(assoc obj alist)
(assp procedure alist)
(assq obj alist)
(assv obj alist)
(atan num)
(atan real₁ real₂)
(begin expr₁ expr₂ ...)
(binary-port? obj)
(bitwise-and exint ...)
(bitwise-arithmetic-shift exint₁ exint₂)
(bitwise-arithmetic-shift-left exint₁ exint₂)
(bitwise-arithmetic-shift-right exint₁ exint₂)
(bitwise-bit-count exint)
(bitwise-bit-field exint₁ exint₂ exint₃)
(bitwise-bit-set? exint₁ exint₂)
(bitwise-copy-bit exint₁ exint₂ exint₃)
(bitwise-copy-bit-field exint₁ exint₂ exint₃ exint₄)
(bitwise-first-bit-set exint)
(bitwise-if exint₁ exint₂ exint₃)
Summary of Forms

(bitwise-ior exint ...) procedure 186
(bitwise-length exint) procedure 187
(bitwise-not exint) procedure 186
(bitwise-reverse-bit-field exint_1 exint_2 exint_3) procedure 186
(bitwise-rotate-bit-field exint_1 exint_2 exint_3 exint_4) procedure 190
(bitwise-xor exint ...) procedure 186
(boolean=? boolean_1 boolean_2) procedure 243
(boolean? obj) procedure 150
(bound-identifier=? identifier_1 identifier_2) procedure 302
(buffer-mode symbol) syntax 261
(buffer-mode? obj) syntax 262
(bytevector->sint-list bytevector eness size) procedure 238
(bytevector->string bytevector transcoder) procedure 286
(bytevector->u8-list bytevector) procedure 232
(bytevector->uint-list bytevector eness size) procedure 238
(bytevector-copy bytevector) procedure 229
(bytevector-copy! src src-start dst dst-start n) procedure 230
(bytevector-fill! bytevector fill) procedure 229
(bytevector-ieee-double-native-ref bytevector n) procedure 239
(bytevector-ieee-double-native-set! bytevector n x) procedure 239
(bytevector-ieee-double-ref bytevector n eness) procedure 240
(bytevector-ieee-double-set! bytevector n x eness) procedure 240
(bytevector-ieee-single-native-ref bytevector n) procedure 239
(bytevector-ieee-single-native-set! bytevector n x) procedure 239
(bytevector-ieee-single-ref bytevector n eness) procedure 240
(bytevector-ieee-single-set! bytevector n x eness) procedure 240
(bytevector-length bytevector) procedure 229
(bytevector-s16-native-ref bytevector n) procedure 232
(bytevector-s16-native-set! bytevector n s16) procedure 233
(bytevector-s16-ref bytevector n eness) procedure 235
(bytevector-s16-set! bytevector n s16 eness) procedure 236
(bytevector-s32-native-ref bytevector n) procedure 232
(bytevector-s32-native-set! bytevector n s32) procedure 233
(bytevector-s32-ref bytevector n eness) procedure 235
(bytevector-s32-set! bytevector n s32 eness) procedure 236
(bytevector-s64-native-ref bytevector n) procedure 232
(bytevector-s64-native-set! bytevector n s64) procedure 233
(bytevector-s64-ref bytevector n eness) procedure 235
(bytevector-s64-set! bytevector n s64 eness) procedure 236
(bytevector-s8-native-ref bytevector n) procedure 231
(bytevector-s8-native-set! bytevector n s8) procedure 231
(bytevector-sint-native-ref bytevector n eness size) procedure 227
(bytevector-sint-native-set! bytevector n sint eness size) procedure 238
(bytevector-u16-native-ref bytevector n) procedure 232
(bytevector-u16-native-set! bytevector n u16) procedure 233
(bytevector-u16-ref bytevector n eness)  procedure 235
(bytevector-u16-set! bytevector n u16 eness)  procedure 236
(bytevector-u32-native-ref bytevector n)  procedure 232
(bytevector-u32-native-set! bytevector n u32)  procedure 233
(bytevector-u32-set! bytevector n u32 eness)  procedure 236
(bytevector-u64-native-ref bytevector n)  procedure 232
(bytevector-u64-native-set! bytevector n u64)  procedure 233
(bytevector-u64-ref bytevector n eness)  procedure 235
(bytevector-u64-set! bytevector n u64 eness)  procedure 236
(bytevector-u8-ref bytevector n)  procedure 230
(bytevector-u8-set! bytevector n u8)  procedure 231
(bytevector-uint-ref bytevector n eness size)  procedure 237
(bytevector-uint-set! bytevector n uint eness size)  procedure 238
(bytevector=? bytevector 1 bytevector 2)  procedure 229
(bytevector? obj)  procedure 155
(caaaar pair)  procedure 156
(caadadr pair)  procedure 157
(caaar pair)  procedure 156
(caadar pair)  procedure 157
(caaddr pair)  procedure 157
(caadr pair)  procedure 157
(caar pair)  procedure 157
(cadaar pair)  procedure 157
(cadadr pair)  procedure 157
(cadar pair)  procedure 157
(caddar pair)  procedure 157
(cadddr pair)  procedure 157
(caddr pair)  procedure 157
(cadr pair)  procedure 157
(call-with-bytevector-output-port procedure)  procedure 266
(call-with-bytevector-output-port procedure ?transcoder)  procedure 266
(call-with-current-continuation procedure)  procedure 123
(call-with-input-file path procedure)  procedure 281
(call-with-output-file path procedure)  procedure 282
(call-with-port port procedure)  procedure 272
(call-with-string-output-port procedure)  procedure 267
(call-with-values producer consumer)  procedure 131
(call/cc procedure)  procedure 123
(car pair)  procedure 156
(case expr clause clause ... )  syntax 113
(case-lambda clause ... )  syntax 94
(cdaaar pair)  procedure 157
(cdaadr pair)  procedure 157
(cdaar pair)  procedure 157
(cdadar pair) procedure 157
(cdaddr pair) procedure 157
(cdadr pair) procedure 157
(cdar pair) procedure 157
(cddaar pair) procedure 157
(cddadr pair) procedure 157
(cddar pair) procedure 157
(cdddar pair) procedure 157
(cdddr pair) procedure 157
(cddr pair) procedure 157
(cdr pair) procedure 156
(ceiling real) procedure 177
(char->integer char) procedure 215
(char-alphabetic? char) procedure 213
(char-ci<=? char1 char2 char3 ...) procedure 212
(char-ci<? char1 char2 char3 ...) procedure 212
(char-ci=? char1 char2 char3 ...) procedure 212
(char-ci>=? char1 char2 char3 ...) procedure 212
(char-ci>? char1 char2 char3 ...) procedure 212
(char-downcase char) procedure 214
(char-foldcase char) procedure 215
(char-general-category char) procedure 214
(char-lower-case? char) procedure 213
(char-numeric? char) procedure 213
(char-title-case? char) procedure 213
(char-titlecase char) procedure 214
(char-upcase char) procedure 214
(char-upper-case? char) procedure 213
(char-whitespace? char) procedure 213
(char<=? char1 char2 char3 ...) procedure 212
(char<? char1 char2 char3 ...) procedure 212
(char=? char1 char2 char3 ...) procedure 212
(char>=? char1 char2 char3 ...) procedure 212
(char>? char1 char2 char3 ...) procedure 212
(char? obj) procedure 154
(close-input-port input-port) procedure 285
(close-output-port output-port) procedure 285
(close-port port) procedure 270
(command-line) procedure 350
(complex? obj) procedure 151
(cond clause1 clause2 ...) syntax 111
(condition condition ...) procedure 362
(condition-accessor rtd procedure) procedure 365
(condition-irritants condition) procedure 368
<table>
<thead>
<tr>
<th>Form</th>
<th>Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(condition-message condition)</code></td>
<td>procedure</td>
<td>368</td>
</tr>
<tr>
<td><code>(condition-predicate rtd)</code></td>
<td>procedure</td>
<td>365</td>
</tr>
<tr>
<td><code>(condition-who condition)</code></td>
<td>procedure</td>
<td>369</td>
</tr>
<tr>
<td><code>(condition? obj)</code></td>
<td>procedure</td>
<td>362</td>
</tr>
<tr>
<td><code>(cons obj₁ obj₂)</code></td>
<td>procedure</td>
<td>156</td>
</tr>
<tr>
<td><code>(cons* obj ... final-obj)</code></td>
<td>procedure</td>
<td>158</td>
</tr>
<tr>
<td><code>constant</code></td>
<td>syntax</td>
<td>141</td>
</tr>
<tr>
<td><code>(cos num)</code></td>
<td>procedure</td>
<td>185</td>
</tr>
<tr>
<td><code>(current-error-port)</code></td>
<td>procedure</td>
<td>263</td>
</tr>
<tr>
<td><code>(current-input-port)</code></td>
<td>procedure</td>
<td>263</td>
</tr>
<tr>
<td><code>(current-output-port)</code></td>
<td>procedure</td>
<td>263</td>
</tr>
<tr>
<td><code>(datum-&gt;syntax template-identifier obj)</code></td>
<td>procedure</td>
<td>308</td>
</tr>
<tr>
<td><code>(define var expr)</code></td>
<td>syntax</td>
<td>100</td>
</tr>
<tr>
<td><code>(define var)</code></td>
<td>syntax</td>
<td>100</td>
</tr>
<tr>
<td><code>(define (var₀ var₁ ...) body₁ body₂ ...)</code></td>
<td>syntax</td>
<td>100</td>
</tr>
<tr>
<td><code>(define (var₀ . var₁) body₁ body₂ ...)</code></td>
<td>syntax</td>
<td>100</td>
</tr>
<tr>
<td><code>(define (var₀ var₁ var₂ ... . varₙ) body₁ body₂ ...)</code></td>
<td>syntax</td>
<td>100</td>
</tr>
<tr>
<td><code>(define-condition-type name parent constructor pred field ...)</code></td>
<td>syntax</td>
<td>364</td>
</tr>
<tr>
<td><code>(define-enumeration name (symbol ...) constructor)</code></td>
<td>syntax</td>
<td>250</td>
</tr>
<tr>
<td><code>(define-record-type record-name clause ...)</code></td>
<td>syntax</td>
<td>328</td>
</tr>
<tr>
<td><code>(define-record-type (record-name constructor pred) clause ...)</code></td>
<td>syntax</td>
<td>328</td>
</tr>
<tr>
<td><code>(define-syntax keyword expr)</code></td>
<td>syntax</td>
<td>292</td>
</tr>
<tr>
<td><code>(delay expr)</code></td>
<td>syntax</td>
<td>128</td>
</tr>
<tr>
<td><code>(delete-file path)</code></td>
<td>procedure</td>
<td>286</td>
</tr>
<tr>
<td><code>(denominator rat)</code></td>
<td>procedure</td>
<td>181</td>
</tr>
<tr>
<td><code>(display obj)</code></td>
<td>procedure</td>
<td>285</td>
</tr>
<tr>
<td><code>(display obj textual-output-port)</code></td>
<td>procedure</td>
<td>285</td>
</tr>
<tr>
<td><code>(div x₁ x₂)</code></td>
<td>procedure</td>
<td>175</td>
</tr>
<tr>
<td><code>(div-and-mod x₁ x₂)</code></td>
<td>procedure</td>
<td>175</td>
</tr>
<tr>
<td><code>(div0 x₁ x₂)</code></td>
<td>procedure</td>
<td>176</td>
</tr>
<tr>
<td><code>(div0-and-mod0 x₁ x₂)</code></td>
<td>procedure</td>
<td>176</td>
</tr>
<tr>
<td><code>(do ((var init update ...) (test result ...) expr ...))</code></td>
<td>syntax</td>
<td>115</td>
</tr>
<tr>
<td><code>(dynamic-wind in body out)</code></td>
<td>syntax</td>
<td>112</td>
</tr>
<tr>
<td><code>(endianness symbol)</code></td>
<td>syntax</td>
<td>228</td>
</tr>
<tr>
<td><code>(enum-set-&gt;list enum-set)</code></td>
<td>procedure</td>
<td>252</td>
</tr>
<tr>
<td><code>(enum-set-complement enum-set)</code></td>
<td>procedure</td>
<td>254</td>
</tr>
<tr>
<td><code>(enum-set-constructor enum-set)</code></td>
<td>procedure</td>
<td>251</td>
</tr>
<tr>
<td><code>(enum-set-difference enum-set₁ enum-set₂)</code></td>
<td>procedure</td>
<td>253</td>
</tr>
<tr>
<td><code>(enum-set-indexer enum-set)</code></td>
<td>procedure</td>
<td>254</td>
</tr>
<tr>
<td><code>(enum-set-intersection enum-set₁ enum-set₂)</code></td>
<td>procedure</td>
<td>253</td>
</tr>
<tr>
<td><code>(enum-set-member? symbol enum-set)</code></td>
<td>procedure</td>
<td>253</td>
</tr>
<tr>
<td><code>(enum-set-projection enum-set₁ enum-set₂)</code></td>
<td>procedure</td>
<td>254</td>
</tr>
<tr>
<td><code>(enum-set-subset? enum-set₁ enum-set₂)</code></td>
<td>procedure</td>
<td>252</td>
</tr>
<tr>
<td><code>(enum-set-union enum-set₁ enum-set₂)</code></td>
<td>procedure</td>
<td>253</td>
</tr>
<tr>
<td>Procedure/Function</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>(enum-set-universe enum-set)</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>(enum-set=? enum-set₁ enum-set₂)</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>(environment import-spec ...)</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>(eof-object)</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>(eof-object? obj)</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>(eol-style symbol)</td>
<td>259</td>
<td></td>
</tr>
<tr>
<td>(eq? obj₁ obj₂)</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>(equal-hash obj)</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>(equal? obj₁ obj₂)</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>(eqv? obj₁ obj₂)</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>(error who msg irritant ...)</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>(error-handling-mode symbol)</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>(error? obj)</td>
<td>367</td>
<td></td>
</tr>
<tr>
<td>(eval obj environment)</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>(even? int)</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>(exact num)</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>(exact-&gt;inexact num)</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>(exact-integer-sqrt n)</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>(exists procedure list₁ list₂ ...)</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>(exit)</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>(exit obj)</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>(exp num)</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>(expt num₁ num₂)</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>(fields)</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>(file-exists? path)</td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>(file-options symbol ...)</td>
<td>261</td>
<td></td>
</tr>
<tr>
<td>(filter procedure list)</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>(find procedure list)</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>(finite? real)</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>(fixnum-&gt;flonum fx)</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>(fixnum-width)</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>(fixnum? obj)</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>(fl* fl ...)</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td>(fl+ fl ...)</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>(fl- fl)</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>(fl- fl₁ fl₂ fl₃ ...)</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>(fl/ fl)</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td>(fl/ fl₁ fl₂ fl₃ ...)</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td>(fl&lt;=? fl₁ fl₂ fl₃ ...)</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>(fl&lt;? fl₁ fl₂ fl₃ ...)</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>(fl=? fl₁ fl₂ fl₃ ...)</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>(fl&gt;=? fl₁ fl₂ fl₃ ...)</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>(fl&gt;? fl₁ fl₂ fl₃ ...)</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>(flabs fl)</td>
<td>209</td>
<td></td>
</tr>
</tbody>
</table>
The Scheme Programming Language, 4th Edition

(flacos fl)
(flasin fl)
(flatan fl)
(flatan fl1 fl2)
(flceiling fl)
(flcos fl)
(fldenominator fl)
(fldiv fl1 fl2)
(fldiv-and-mod fl1 fl2)
(fldiv0 fl1 fl2)
(fldiv0-and-mod0 fl1 fl2)
(fleven? fl-int)
(flexp fl)
(flexpt fl1 fl2)
(flfinite? fl)
(flfloor fl)
(flfinite? fl)
(flinteger? fl)
(fllog fl)
(fllog fl1 fl2)
(flmax fl1 fl2 ...)
(flmin fl1 fl2 ...)
(flmod fl1 fl2)
(flmod0 fl1 fl2)
(flnan? fl)
(flnegative? fl)
(flnumerator fl)
(flodd? fl-int)
(flonum? obj)
(floor real)
(flpositive? fl)
(flround fl)
(flsin fl)
(flsqrt fl)
(fltan fl)
(fltruncate fl)
(flush-output-port output-port)
(flzero? fl)
(fold-left procedure obj list1 list2 ...)
(fold-right procedure obj list1 list2 ...)
(for-all procedure list1 list2 ...)
(for-each procedure list1 list2 ...)
(force promise)
(free-identifier=? identifier1 identifier2)
(fx* fx1 fx2)
### Summary of Forms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fx* carry fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fx+ fx₁ fx₂)</td>
<td>Procedure</td>
<td>195</td>
</tr>
<tr>
<td>(fx+/carry fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fx- fx)</td>
<td>Procedure</td>
<td>195</td>
</tr>
<tr>
<td>(fx- fx₁ fx₂)</td>
<td>Procedure</td>
<td>195</td>
</tr>
<tr>
<td>(fx-/carry fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fx&lt;=? fx₁ fx₂ fx₃ ...)</td>
<td>Procedure</td>
<td>193</td>
</tr>
<tr>
<td>(fx=? fx₁ fx₂ fx₃ ...)</td>
<td>Procedure</td>
<td>193</td>
</tr>
<tr>
<td>(fx&gt;=? fx₁ fx₂ fx₃ ...)</td>
<td>Procedure</td>
<td>193</td>
</tr>
<tr>
<td>(fx&gt;? fx₁ fx₂ fx₃ ...)</td>
<td>Procedure</td>
<td>193</td>
</tr>
<tr>
<td>(fxand fx ...)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fxarithmetic-shift fx₁ fx₂)</td>
<td>Procedure</td>
<td>201</td>
</tr>
<tr>
<td>(fxarithmetic-shift-left fx₁ fx₂)</td>
<td>Procedure</td>
<td>201</td>
</tr>
<tr>
<td>(fxarithmetic-shift-right fx₁ fx₂)</td>
<td>Procedure</td>
<td>201</td>
</tr>
<tr>
<td>(fxbit-count fx)</td>
<td>Procedure</td>
<td>198</td>
</tr>
<tr>
<td>(fxbit-field fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>200</td>
</tr>
<tr>
<td>(fxbit-set? fx₁ fx₂)</td>
<td>Procedure</td>
<td>199</td>
</tr>
<tr>
<td>(fxcopy-bit fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>200</td>
</tr>
<tr>
<td>(fxcopy-bit-field fx₁ fx₂ fx₃ fx₄)</td>
<td>Procedure</td>
<td>200</td>
</tr>
<tr>
<td>(fxdiv fx₁ fx₂)</td>
<td>Procedure</td>
<td>196</td>
</tr>
<tr>
<td>(fxdiv-and-mod fx₁ fx₂)</td>
<td>Procedure</td>
<td>196</td>
</tr>
<tr>
<td>(fxdiv0 fx₁ fx₂)</td>
<td>Procedure</td>
<td>196</td>
</tr>
<tr>
<td>(fxdiv0-and-mod0 fx₁ fx₂)</td>
<td>Procedure</td>
<td>196</td>
</tr>
<tr>
<td>(fxeven? fx)</td>
<td>Procedure</td>
<td>194</td>
</tr>
<tr>
<td>(fxfirst-bit-set fx)</td>
<td>Procedure</td>
<td>199</td>
</tr>
<tr>
<td>(fxif fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>198</td>
</tr>
<tr>
<td>(fxior fx ...)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fxlength fx)</td>
<td>Procedure</td>
<td>198</td>
</tr>
<tr>
<td>(fxmax fx₁ fx₂ ...)</td>
<td>Procedure</td>
<td>195</td>
</tr>
<tr>
<td>(fxmin fx₁ fx₂ ...)</td>
<td>Procedure</td>
<td>195</td>
</tr>
<tr>
<td>(fxmod fx₁ fx₂)</td>
<td>Procedure</td>
<td>196</td>
</tr>
<tr>
<td>(fxmod0 fx₁ fx₂)</td>
<td>Procedure</td>
<td>196</td>
</tr>
<tr>
<td>(fxnegative? fx)</td>
<td>Procedure</td>
<td>194</td>
</tr>
<tr>
<td>(fxnot fx)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fxodd? fx)</td>
<td>Procedure</td>
<td>194</td>
</tr>
<tr>
<td>(fxpositive? fx)</td>
<td>Procedure</td>
<td>194</td>
</tr>
<tr>
<td>(fxreverse-bit-field fx₁ fx₂ fx₃)</td>
<td>Procedure</td>
<td>202</td>
</tr>
<tr>
<td>(fxrotate-bit-field fx₁ fx₂ fx₃ fx₄)</td>
<td>Procedure</td>
<td>201</td>
</tr>
<tr>
<td>(fxxor fx ...)</td>
<td>Procedure</td>
<td>197</td>
</tr>
<tr>
<td>(fxzero? fx)</td>
<td>Procedure</td>
<td>194</td>
</tr>
<tr>
<td>(gcd int ...)</td>
<td>Procedure</td>
<td>179</td>
</tr>
<tr>
<td>(generate-temporaries list)</td>
<td>Procedure</td>
<td>310</td>
</tr>
<tr>
<td>(get-bytevector-all binary-input-port)</td>
<td>Procedure</td>
<td>275</td>
</tr>
<tr>
<td>(get-bytevector-n binary-input-port n)</td>
<td>Procedure</td>
<td>274</td>
</tr>
</tbody>
</table>
The Scheme Programming Language, 4th Edition

(get-bytevector-n! binary-input-port bytevector start n)  procedure 274
(get-bytevector-some binary-input-port)  procedure 275
(get-char textual-input-port)  procedure 275
(get-datum textual-input-port)  procedure 278
(get-line textual-input-port)  procedure 277
(get-string-all textual-input-port)  procedure 277
(get-string-n textual-input-port n)  procedure 276
(get-string-n! textual-input-port string start n)  procedure 276
(get-u8 binary-input-port)  procedure 274
(greatest-fixnum)  procedure 193
(guard (var clause1 clause2 ...) b1 b2 ...)  syntax 361
(hashtable-clear! hashtable)  procedure 249
(hashtable-clear! hashtable size)  procedure 249
(hashtable-contains? hashtable key)  procedure 246
(hashtable-copy hashtable)  procedure 248
(hashtable-copy hashtable mutable?)  procedure 248
(hashtable-delete! hashtable key)  procedure 248
(hashtable-entries hashtable)  procedure 250
(hashtable-equivalence-function hashtable)  procedure 245
(hashtable-hash-function hashtable)  procedure 245
(hashtable-keys hashtable)  procedure 249
(hashtable-mutable? hashtable)  procedure 245
(hashtable-ref hashtable key default)  procedure 246
(hashtable-set! hashtable key obj)  procedure 246
(hashtable-size hashtable)  procedure 248
(hashtable-update! hashtable key procedure default)  procedure 247
(hashtable? obj)  procedure 155
(i/o-decoding-error? obj)  procedure 375
(i/o-encoding-error-char condition)  procedure 376
(i/o-encoding-error? obj)  procedure 376
(i/o-error-filename condition)  procedure 373
(i/o-error-port condition)  procedure 375
(i/o-error-position condition)  procedure 372
(i/o-error? obj)  procedure 371
(i/o-file-already-exists-error? obj)  procedure 374
(i/o-file-does-not-exist-error? obj)  procedure 374
(i/o-file-is-read-only-error? obj)  procedure 374
(i/o-file-protection-error? obj)  procedure 373
(i/o-filename-error? obj)  procedure 373
(i/o-invalid-position-error? obj)  procedure 372
(i/o-port-error? obj)  procedure 375
(i/o-read-error? obj)  procedure 372
(i/o-write-error? obj)  procedure 372
(identifier-syntax tmpl)  syntax 297
(identifier-syntax (id1 tmpl1) ((set! id2 e2) tmpl2))  syntax 297
The Scheme Programming Language, 4th Edition

Summary of Forms

(identifier? obj)  procedure 301
(if test consequent alternative)  syntax 109
(if test consequent)  syntax 109
(imag-part num)  procedure 182
immutable

(implementation-restriction-violation? obj)  procedure 369
(inexact num)  procedure 180
(inexact->exact num)  procedure 181
(inexact? num)  procedure 170
(infinite? real)  procedure 174
(input-port? obj)  procedure 270
(integer->char n)  procedure 215
(integer-valued? obj)  procedure 153
(integer? obj)  procedure 151
(irritants-condition? obj)  procedure 368
(lambda formals body1 body2 ...)  syntax 92
(latin-1-codec)  procedure 259
(lcm int ...)  procedure 179

(length list)  procedure 193

(let ((var expr) ...) body1 body2 ...)  syntax 95
(let name ((var expr) ...) body1 body2 ...)  syntax 114
(let* ((var expr) ...) body1 body2 ...)  syntax 96
(let*-values ((formals expr) ...) body1 body2 ...)  syntax 99
(let-syntax ((keyword expr) ...) form1 form2 ...)  syntax 293
(let-values ((formals expr) ...) body1 body2 ...)  syntax 99
(letrec ((var expr) ...) body1 body2 ...)  syntax 97
(letrec* ((var expr) ...) body1 body2 ...)  syntax 98
(letrec-sysyntax ((keyword expr) ...) form1 form2 ...)  syntax 293
(lexical-violation? obj)  procedure 370
(list obj ...)  procedure 158
(list->string list)  procedure 223
(list->vector list)  procedure 226
(list-ref list n)  procedure 159
(list-sort predicate list)  procedure 167
(list-tail list n)  procedure 160
(list? obj)  procedure 158
(log num)  procedure 184
(log num1 num2)  procedure 184
(lookahead-char textual-input-port)  procedure 275
(lookahead-u8 binary-input-port)  procedure 274
(magnitude num)  procedure 183
(make-assertion-violation)  procedure 366
(make-bytevector n)  procedure 228
(make-bytevector n fill)  procedure 228
(make-custom-binary-input-port id r! gp sp! close)
(procedure 267)
(make-custom-binary-input/output-port id r! w! gp sp! close)
(procedure 267)
(make-custom-binary-output-port id w! gp sp! close)
(procedure 267)
(make-custom-textual-input-port id r! gp sp! close)
(procedure 268)
(make-custom-textual-input/output-port id r! w! gp sp! close)
(procedure 268)
(make-custom-textual-output-port id w! gp sp! close)
(procedure 268)
(make-enumeration symbol-list)
(procedure 251)
(make-eq-hashtable)
(procedure 243)
(make-eq-hashtable size)
(procedure 243)
(make-eqv-hashtable)
(procedure 244)
(make-eqv-hashtable size)
(procedure 244)
(make-error)
(procedure 367)
(make-hashtable hash equiv?)
(procedure 244)
(make-hashtable hash equiv? size)
(procedure 244)
(make-i/o-decoding-error pobj)
(procedure 375)
(make-i/o-encoding-error pobj obj)
(procedure 376)
(make-i/o-error)
(procedure 371)
(make-i/o-file-already-exists-error filename)
(procedure 374)
(make-i/o-file-does-not-exist-error filename)
(procedure 374)
(make-i/o-file-is-read-only-error filename)
(procedure 374)
(make-i/o-file-protection-error filename)
(procedure 373)
(make-i/o-file-name-error filename)
(procedure 373)
(make-i/o-invalid-position-error position)
(procedure 372)
(make-i/o-port-error pobj)
(procedure 375)
(make-i/o-read-error)
(procedure 372)
(make-i/o-write-error)
(procedure 372)
(make-implementation-restriction-violation)
(procedure 369)
(make-irritants-condition irritants)
(procedure 368)
(make-lexical-violation)
(procedure 370)
(make-message-condition message)
(procedure 368)
(make-no-infinities-violation)
(procedure 376)
(make-no-nans-violation)
(procedure 377)
(make-non-continuable-violation)
(procedure 369)
(make-polar real₁ real₂)
(procedure 183)
(make-record-constructor-descriptor rtd parent-rcd protocol)
(procedure 332)
(make-record-type-descriptor name parent uid s? o? fields)
(procedure 331)
(make-rectangular real₁ real₂)
(procedure 182)
(make-serious-condition)
(procedure 366)
(make-string n)
(procedure 218)
(make-string n char)
(procedure 218)
(make-syntax-violation form subform)
(procedure 370)
(make-transcoder codec)
(procedure 259)
(make-transcoder codec eol-style)
(procedure 259)
(make-transcoder codec eol-style error-handling-mode)
(procedure 259)
(make-undefined-violation)
(procedure 371)
The Scheme Programming Language, 4th Edition

Summary of Forms

<table>
<thead>
<tr>
<th>Form</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>make-variable-transformer procedure</code></td>
<td>306</td>
</tr>
<tr>
<td><code>make-vector n</code></td>
<td>224</td>
</tr>
<tr>
<td><code>make-vector n obj</code></td>
<td>224</td>
</tr>
<tr>
<td><code>make-violation</code></td>
<td>366</td>
</tr>
<tr>
<td><code>make-warning</code></td>
<td>367</td>
</tr>
<tr>
<td><code>make-who-condition who</code></td>
<td>369</td>
</tr>
<tr>
<td><code>map procedure list1 list2 ...</code></td>
<td>117</td>
</tr>
<tr>
<td><code>max real1 real2 ...</code></td>
<td>178</td>
</tr>
<tr>
<td><code>member obj list</code></td>
<td>161</td>
</tr>
<tr>
<td><code>memp procedure list</code></td>
<td>163</td>
</tr>
<tr>
<td><code>memq obj list</code></td>
<td>161</td>
</tr>
<tr>
<td><code>memv obj list</code></td>
<td>161</td>
</tr>
<tr>
<td><code>message-condition? obj</code></td>
<td>368</td>
</tr>
<tr>
<td><code>min real1 real2 ...</code></td>
<td>178</td>
</tr>
<tr>
<td><code>mod x1 x2</code></td>
<td>175</td>
</tr>
<tr>
<td><code>mod0 x1 x2</code></td>
<td>176</td>
</tr>
<tr>
<td><code>modulo int1 int2</code></td>
<td>175</td>
</tr>
<tr>
<td><code>mutable</code></td>
<td>331</td>
</tr>
<tr>
<td><code>nan? real</code></td>
<td>174</td>
</tr>
<tr>
<td><code>nativeendianness</code></td>
<td>228</td>
</tr>
<tr>
<td><code>native-eol-style</code></td>
<td>260</td>
</tr>
<tr>
<td><code>native-transcoder</code></td>
<td>259</td>
</tr>
<tr>
<td><code>negative? real</code></td>
<td>173</td>
</tr>
<tr>
<td><code>newline</code></td>
<td>285</td>
</tr>
<tr>
<td><code>newline textual-output-port</code></td>
<td>285</td>
</tr>
<tr>
<td><code>no-infinities-violation? obj</code></td>
<td>376</td>
</tr>
<tr>
<td><code>no-nans-violation? obj</code></td>
<td>377</td>
</tr>
<tr>
<td><code>non-continuable-violation? obj</code></td>
<td>369</td>
</tr>
<tr>
<td><code>nongenerative</code></td>
<td>331</td>
</tr>
<tr>
<td><code>not obj</code></td>
<td>110</td>
</tr>
<tr>
<td><code>null-environment version</code></td>
<td>137</td>
</tr>
<tr>
<td><code>null? obj</code></td>
<td>151</td>
</tr>
<tr>
<td><code>number-&gt;string num</code></td>
<td>191</td>
</tr>
<tr>
<td><code>number-&gt;string num radix</code></td>
<td>191</td>
</tr>
<tr>
<td><code>number-&gt;string num radix precision</code></td>
<td>191</td>
</tr>
<tr>
<td><code>number? obj</code></td>
<td>151</td>
</tr>
<tr>
<td><code>numerator rat</code></td>
<td>181</td>
</tr>
<tr>
<td><code>odd? int</code></td>
<td>174</td>
</tr>
<tr>
<td><code>opaque</code></td>
<td>331</td>
</tr>
<tr>
<td><code>open-bytevector-input-port bytevector</code></td>
<td>264</td>
</tr>
<tr>
<td><code>open-bytevector-input-port bytevector ?transcoder</code></td>
<td>264</td>
</tr>
<tr>
<td><code>open-bytevector-output-port</code></td>
<td>265</td>
</tr>
<tr>
<td><code>open-bytevector-output-port ?transcoder</code></td>
<td>265</td>
</tr>
<tr>
<td><code>open-file-input-port path</code></td>
<td>262</td>
</tr>
<tr>
<td><code>open-file-input-port path options</code></td>
<td>262</td>
</tr>
</tbody>
</table>
(open-file-input-port path options b-mode)  procedure 262
(open-file-input-port path options b-mode ?transcoder)  procedure 262
(open-file-input/output-port path)  procedure 263
(open-file-input/output-port path options)  procedure 263
(open-file-input/output-port path options b-mode)  procedure 263
(open-file-input/output-port path options b-mode ?transcoder)  procedure 263
(open-file-output-port path)  procedure 262
(open-file-output-port path options)  procedure 262
(open-file-output-port path options b-mode)  procedure 262
(open-file-output-port path options b-mode ?transcoder)  procedure 262
(open-input-file path)  procedure 280
(open-output-file path)  procedure 281
(open-string-input-port string)  procedure 265
(open-string-output-port)  procedure 266
(or expr ...)  syntax 110
(output-port-buffer-mode port)  procedure 273
(output-port? obj)  procedure 270
(pair? obj)  procedure 151
parent  syntax 331
parent-rtd  syntax 331
(partition procedure list)  procedure 164
(peek-char)  procedure 284
(peek-char textual-input-port)  procedure 284
(port-eof? input-port)  procedure 278
(port-has-port-position? port)  procedure 271
(port-has-set-port-position? port)  procedure 272
(port-position port)  procedure 271
(port-transcoder port)  procedure 271
(port? obj)  procedure 270
(positive? real)  procedure 173
(expr₀ expr₁ ...)  syntax 107
(procedure? obj)  procedure 155
protocol  syntax 331
(put-bytevector binary-output-port bytevector)  procedure 279
(put-bytevector binary-output-port bytevector start)  procedure 279
(put-bytevector binary-output-port bytevector start n)  procedure 279
(put-char textual-output-port char)  procedure 279
(put-datum textual-output-port obj)  procedure 279
(put-string textual-output-port string)  procedure 279
(put-string textual-output-port string start)  procedure 279
(put-string textual-output-port string start n)  procedure 279
(put-u8 binary-output-port octet)  procedure 278
(quasiquote obj ...)  syntax 142
(quasisyntax template ...)  syntax 305
(quote obj)  syntax 141
(quotient int₁ int₂)
(raise obj)
(raise-continuable obj)
(rational-valued? obj)
(rational? obj)
(rationalize real₁ real₂)
(read)
(read textual-input-port)
(read-char)
(read-char textual-input-port)
(real->flonum real)
(real-part num)
(real-valued? obj)
(real? obj)
(record-accessor rtd idx)
(record-constructor rcd)
(record-constructor-descriptor record-name)
(record-field-mutable? rtd idx)
(record-mutator rtd idx)
(record-predicate rtd)
(record-rtd record)
(record-type-descriptor record-name)
(record-type-descriptor? obj)
(record-type-field-names rtd)
(record-type-generative? rtd)
(record-type-name rtd)
(record-type-opaque? rtd)
(record-type-parent rtd)
(record-type-sealed? rtd)
(record-type-uid rtd)
(record? obj)
(remainder int₁ int₂)
(remove obj list)
(remp procedure list)
(remq obj list)
(remv obj list)
(reverse list)
(round real)
(scheme-report-environment version)
sealed
(serious-condition? obj)
(set! var expr)
(set-car! pair obj)
(set-cdr! pair obj)
(set-port-position! port pos)
(simple-conditions condition)
  procedure 363
(sin num)
  procedure 185
(sint-list->bytevector list eness size)
  procedure 239
(sqrt num)
  procedure 183
(standard-error-port)
  procedure 264
(standard-input-port)
  procedure 264
(standard-output-port)
  procedure 264
(string char ...)
  procedure 218
(string->bytevector string transcoder)
  procedure 287
(string->list string)
  procedure 222
(string->number string)
  procedure 191
(string->number string radix)
  procedure 191
(string->symbol string)
  procedure 242
(string->utf16 string)
  procedure 287
(string->utf16 string endianness)
  procedure 287
(string->utf32 string)
  procedure 287
(string->utf32 string endianness)
  procedure 287
(string->utf8 string)
  procedure 287
(string-append string ...)
  procedure 219
(string-ci-hash string)
  procedure 245
(string-ci<=? string1 string2 string3 ...)
  procedure 217
(string-ci<? string1 string2 string3 ...)
  procedure 217
(string-ci=? string1 string2 string3 ...)
  procedure 217
(string-ci=>= string1 string2 string3 ...)
  procedure 217
(string-ci>? string1 string2 string3 ...)
  procedure 217
(string-copy string)
  procedure 219
(string-downcase string)
  procedure 221
(string-fill! string char)
  procedure 220
(string-foldcase string)
  procedure 221
(string-for-each procedure string1 string2 ...)
  procedure 122
(string-hash string)
  procedure 245
(string-length string)
  procedure 218
(string-normalize-nfc string)
  procedure 222
(string-normalize-nfd string)
  procedure 222
(string-normalize-nfkc string)
  procedure 222
(string-normalize-nfkd string)
  procedure 222
(string-ref string n)
  procedure 218
(string-set! string n char)
  procedure 219
(string-titlecase string)
  procedure 221
(string-upcase string)
  procedure 221
(string<=? string1 string2 string3 ...)
  procedure 216
(string<? string1 string2 string3 ...)
  procedure 216
(string=? string1 string2 string3 ...)
  procedure 216
(string=>= string1 string2 string3 ...)
  procedure 216
(string>? string1 string2 string3 ...)
  procedure 216

Summary of Forms
The Scheme Programming Language, 4th Edition

Summary of Forms

(string? obj) procedure 154
(substring string start end) procedure 220
(symbol->string symbol) procedure 242
(symbol-hash symbol) procedure 245
(symbol=? symbol1 symbol2) procedure 242
(symbol? obj) procedure 154
(syntax template) syntax 300
(syntax->datum obj) procedure 308
(syntax-case expr (literal ...) clause ...) syntax 299
(syntax-rules (literal ...) clause ...) syntax 294
(syntax-violation who msg form) procedure 359
(syntax-violation who msg form subform) procedure 359
(syntax-violation-form condition) procedure 370
(syntax-violation-subform condition) procedure 370
(syntax-violation? obj) procedure 370
(tan num) procedure 185
(textual-port? obj) procedure 270
(transcoded-port binary-port transcoder) procedure 271
(transcoder-codec transcoder) procedure 259
(transcoder-eol-style transcoder) procedure 259
(transcoder-error-handling-mode transcoder) procedure 259
(truncate real) procedure 177
(u8-list->bytevector list) procedure 232
(uint-list->bytevector list eness size) procedure 239
(undefined-violation? obj) procedure 371
(unless test-expr expr1 expr2 ...) syntax 112
(unquote obj ...) syntax 142
(unquote-splicing obj ...) syntax 142
(unsyntax template ...) syntax 305
(unsyntax-splicing template ...) syntax 305
(utf-16-codec) procedure 259
(utf-8-codec) procedure 259
(utf16->string bytevector endianness) procedure 288
(utf16->string bytevector endianness endianness-mandatory?) procedure 288
(utf32->string bytevector endianness) procedure 288
(utf32->string bytevector endianness-endianness-mandatory?) procedure 288
(utf8->string bytevector) procedure 287
(values obj ...) syntax 91
(variable
(vector obj ...) procedure 224
(vector->list vector) procedure 225
(vector-fill! vector obj) procedure 225
(vector-for-each procedure vector1 vector2 ...) procedure 122
(vector-length vector) procedure 224
(vector-map procedure vector1 vector1 ...) procedure 121
(vector-ref vector n)  procedure 224
(vector-set! vector n obj)  procedure 225
(vector-sort predicate vector)  procedure 226
(vector-sort! predicate vector)  procedure 226
(vector? obj)  procedure 154
(violation? obj)  procedure 366
(warning? obj)  procedure 367
(when test-expr expr₁ expr₂ ...)  syntax 112
(who-condition? obj)  procedure 369
(with-exception-handler procedure thunk)  procedure 360
(with-input-from-file path thunk)  procedure 283
(with-output-to-file path thunk)  procedure 283
(with-syntax ((pattern expr) ...) body₁ body₂ ...)  syntax 304
(write obj)  procedure 284
(write obj textual-output-port)  procedure 284
(write-char char)  procedure 285
(write-char char textual-output-port)  procedure 285
(zero? num)  procedure 173

R. Kent Dybvig / The Scheme Programming Language, Fourth Edition
Copyright © 2009 The MIT Press. Electronically reproduced by permission.
Illustrations © 2009 Jean-Pierre Hébert
ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
to order this book / about this book

http://www.scheme.com

The Scheme Programming Language, 4th Edition

Written by R. Kent Dybvig

Illustrations by Jean-Pierre Hébert

From the back cover

Scheme is a general-purpose programming language, descended from Algol and Lisp, widely used in computing education and research and a broad range of industrial applications. This thoroughly updated edition of The Scheme Programming Language provides an introduction to Scheme and a definitive reference for standard Scheme, presented in a clear and concise manner. Written for professionals and students with some prior programming experience, it begins by leading the programmer gently through the basics of Scheme and continues with an introduction to some of the more advanced features of the language.

The fourth edition has been substantially revised and expanded to bring the content up to date with the current Scheme standard, the Revised6 Report on Scheme. All parts of the book were updated and three new chapters were added, covering the language’s new library, exception handling, and record-definition features.

The book offers three chapters of introductory material with numerous examples, eight chapters of reference material, and one chapter of extended examples and additional exercises. All of the examples can be entered directly from the keyboard into an interactive Scheme session. Answers to many of the exercises, a complete formal syntax of Scheme, and a summary of forms and procedures are provided in appendixes.

372
The Scheme Programming Language is the only book available that serves both as an introductory text in a variety of courses and as an essential reference for Scheme programmers.

About the Author

R. Kent Dybvig is Professor of Computer Science at Indiana University and principal developer of Chez Scheme.

Endorsements

"Kent Dybvig's The Scheme Programming Language is to Scheme what Kernighan and Ritchie's The C Programming Language is to C. Kent's book is the book for either the novice or serious Scheme programmer. Its style, wit, and organization has reached a new high with the publication of the fourth edition."
—Daniel P. Friedman, Department of Computer Science, Indiana University

"Students in my Programming Language Concepts class need to learn the basics of Scheme in a few days, and to pick up harder concepts throughout the course. For 19 years, The Scheme Programming Language has been an excellent guide for them. Dybvig's rapid-fire prose and examples serve both the Scheme beginner and the experienced programmer in need of a reference. Seldom do my students make a point of praising a computer science textbook; that happens over and over with this one."
—Claude W. Anderson, Rose-Hulman Institute of Technology

"Eric Raymond once wrote that learning Lisp makes one a better programmer for the rest of one's days. Scheme is the best dialect of Lisp to learn for this purpose, and Kent Dybvig's book provides a comprehensive and beautiful introduction to learning Scheme and becoming a better programmer."
—Olivier Danvy, Aarhus University, Denmark, co-Editor-in-Chief of Higher-Order and Symbolic Computation
Index

All page numbers appearing here refer to the printed version of this book and also serve as hypertext links to the corresponding locations in the electronic version of this book.

! (exclamation point), 8
" (double quote), 216
#:r6rs, 456
#' (syntax), 300
#, (unsyntax), 305
#:e (unsyntax-splicing), 305
#; (datum comment), 455
#:\, 211
#' (quasisyntax), 305
#|...|# (block comment), 455
#b (binary), 169
#d (decimal), 169
#:r, 7, 36, 143
#:o (octal), 169
#:t, 7, 36, 143
#:x (hexadecimal), 169
&assertion, 366
&condition, 362
&error, 367
&i/o, 371
&i/o-decoding, 375
&i/o-encoding, 376
&i/o-file-already-exists, 374
&i/o-file-does-not-exist, 374
&i/o-file-is-read-only, 374
&i/o-file-protection, 373
&i/o-filename, 373
&i/o-invalid-position, 372
&i/o-port, 375
&i/o-read, 372
&i/o-write, 372
&implementation-restriction, 369
&irritants, 368
&lexical, 370
&message, 368
&no-infinities, 376
&no-nans, 377
&non-continuable, 369
&serious, 366
&syntax, 370
&undefined, 371
&violation, 366
&warning, 367
&who, 369
'(quote), 17, 22, 59, 141
( ), 7, 19
*, 16, 172
+, 16, 171
, (unquote), 142
bitwise-bit-set?, 188
bitwise-copy-bit, 188
bitwise-copy-bit-field, 189
bitwise-first-bit-set, 187
bitwise-if, 186
bitwise-ior, 186
bitwise-length, 187
bitwise-not, 186
bitwise-reverse-bit-field, 191
bitwise-rotate-bit-field, 190
bitwise-xor, 186
block buffering, 258
block comment (#|...|#), 455
block structure, 4
boolean syntax, 457
boolean values, 7
boolean=?, 243
boolean?, 150
bound-identifier=?, 302
brackets ([ ]), 7, 155
break, 308
buffer modes, 258
buffer-mode, 261
buffer-mode?, 262
bytevector syntax, 461
bytevector->sint-list, 238
bytevector->string, 286
bytevector->u8-list, 232
bytevector->uint-list, 238
bytevector-copy, 229
bytevector-copy!, 230
bytevector-fill!, 229
bytevector-ieee-double-native-ref, 239
bytevector-ieee-double-native-set!, 239
bytevector-ieee-double-ref, 240
bytevector-ieee-double-set!, 240
bytevector-ieee-single-native-ref, 239
bytevector-ieee-single-native-set!, 239
bytevector-ieee-single-ref, 240
bytevector-ieee-single-set!, 240
bytevector-length, 229
bytevector-s16-native-ref, 232
bytevector-s16-native-set!, 233
bytevector-s16-ref, 235
bytevector-s16-set!, 236
bytevector-s32-native-ref, 232
bytevector-s32-native-set!, 233
bytevector-s32-ref, 235
bytevector-s32-set!, 236
bytevector-s64-native-ref, 232
bytevector-s64-native-set!, 233
bytevector-s64-ref, 235
bytevector-s64-set!, 236
bytevector-s8-ref, 231
bytevector-s8-set!, 231
bytevector-sint-ref, 237
bytevector-sint-set!, 238
bytevector-u16-native-ref, 232
bytevector-u16-native-set!, 233
bytevector-u16-ref, 235
bytevector-u16-set!, 236
bytevector-u32-native-ref, 232
bytevector-u32-native-set!, 233
bytevector-u32-ref, 235
bytevector-u32-set!, 236
bytevector-u64-native-ref, 232
bytevector-u64-native-set!, 233
bytevector-u64-ref, 235
bytevector-u64-set!, 236
bytevector-u8-ref, 230
bytevector-u8-set!, 231
bytevector-uint-ref, 237
bytevector-uint-set!, 238
bytevector=, 229
bytevector?, 155
C, 393
caaaaar, 157
caaadr, 157
caar, 157
caadar, 157
caaadr, 157
caaar, 157
cadr, 31, 32, 34, 157
cadadr, 157
cadar, 157
cadddr, 157
caddar, 157
caddr, 157
caddr, 34
call-by-name, 408
call-by-value, 407
call-with-bytevector-output-port, 266
call-with-current-continuation, 123, 426
call-with-input-file, 281
call-with-output-file, 282
call-with-port, 272
call-with-string-output-port, 267
call-with-values, 130, 131
call/cc, 74, 122, 123, 126, 133, 425, 426
car, 18, 155, 156
case, 55, 113, 306
case-lambda, 94
cdaaar, 157
cdaadr, 157
cdaar, 157
cdadar, 157
Index

The Scheme Programming Language, 4th Edition

cdaddr, 157
cdadr, 157
cdar, 157
cddaar, 157
cddadr, 157
cddar, 157
cdddar, 157
cddddr, 157
cdddr, 157
cddr, 31, 34, 157
cdr, 18, 38, 155, 156
ceiling, 177
char->integer, 215
char-alphabetic?, 213
char-ci<=?, 212
char-ci<?, 212
char-ci=?, 212
char-ci>=?, 212
char-ci>?, 212
char-downcase, 214
char-foldcase, 215
char-general-category, 214
char-lower-case?, 213
char-numeric?, 213
char-title-case?, 213
char-titlecase, 214
char-upcase, 214
char-upper-case?, 213
char-whitespace?, 213
char<=?, 212
char<?, 212
char=?, 212
char>??, 212
char>=?, 212
char>?, 212
char?, 154
character syntax, 457
characters, 211
Chez Scheme, ix, 42
child type, 325
circular lists, 156
close-input-port, 285
close-output-port, 285
close-port, 270
codec, 257
command-line, 350
comments, 7, 455
Common Lisp, 6
compiler, 4
complete, see engines
complex numbers, 167, 412
complex?, 151, 167
compose, 34
compound condition, 362
cond, 39, 44, 111, 304
condition, 362
condition object, 361
condition type, 361
condition-accessor, 365
condition-irritants, 368
condition-message, 368
condition-predicate, 365
condition-who, 369
condition?, 362
conditionals, 109
conditions, 357
cons, 19, 156
cons cell, 155
cons*, 158
consing, 19
constant, 141
constants, 21, 141
continuation-passing style, 78, 418
continuations, 5, 73, 124, 421
control structures, 107
core syntactic forms, 4, 22, 59, 404
cos, 185
CPS, 78
current exception handler, 357
current-error-port, 263
current-input-port, 263
current-output-port, 263
cyclic lists, 56
d (double), 169
data, 141
datum comment ( #; ), 455
datum syntax, 455, 456
datum->syntax, 308, 317, 320
default protocol, 327
define, 30, 81, 100
define-condition-type, 364
define-enumeration, 250
define-integrable, 315
define-object, 408
define-record-type, 323, 328
define-structure, 318
define-syntax, 61, 291, 292, 389
defining syntactic extensions, 60
defun syntax, 33, 60
delay, 128
delayed evaluation, 408
delete-file, 286
delq!, 54
denominator, 181
describe-segment, 132
display, 285, 397
div, 175
div-and-mod, 175
div0, 176
div0-and-mod0, 176
divisors, 115, 116
do, 115, 312
dot ( . ), 19, 460
dotted pair, 20, 155
double, 27, 33
double quotes, 216
double-any, 30
double-cons, 27, 33
doubler, 33
doubly recursive, 70
dx, 131
dynamic allocation, 3
dynamic-wind, 124
ellipsis ( . . . ), 61, 294
else, 111, 112, 113
empty list, 7, 19
endianness, 228
engines, 421
enum-set->list, 252
enum-set-complement, 254
enum-set-constructor, 251
enum-set-difference, 257
enum-set-indexer, 254
enum-set-intersection, 253
enum-set-member?, 253
enum-set-projection, 254
enum-set-subset?, 254
enum-set-union, 253
enum-set-universe, 252
enum-set=?, 252
environment, 137
environment, 404
eof object, 257
eof-object, 273
eof-object?, 257, 273
eol style, 257
eol-style, 259
eq?, 143
equal-hash, 245
equal?, 148
equivalence predicates, 143
eqv?, 38, 146
error, 358
error handling mode, 258
error-handling-mode, 260
error?, 367
eval, 136
even?, 47, 66, 81, 174
exact, 180
exact->inexact, 181
exact-integer-sqrt, 184
exact?, 167, 170
exactness, 167, 180
exactness preserving, 167
except import set, 346
exceptions, 9, 357
exclamation point ( ! ), 8
exists, 119
exit, 350
exp, 184
expansion, 59
expire, see engines
export, 345
export level, 345
expressions, 7
expt, 179
extended examples, 381
f (single), 169
factor, 71, 72, 73
factorial, 68, 73, 116
false, 7, 36
fast Fourier transform (FFT), 412
fenders, 299, 301
fibonacci, 69, 102, 116, 422
Fibonacci numbers, 69, 102
fields, 331
file, 257
file-exists?, 286
file-options, 261
filter, 164
find, 165
finite?, 174
first-class data values, 3
first-class procedures, 5
fixnum, 192
fixnum->flonum, 211
fixnum-width, 193
fixnum?, 193
fl*, 207
fl+, 206
fl-, 206
fl/, 207
fl<=?, 203
fl<?, 203
fl=?, 203
fl>=?, 203
fl>?, 203
flabs, 209
flacos, 210
flasin, 210
flatan, 210
flceiling, 208
flcos, 210
fldenominator, 209
fldiv, 207
fldiv-and-mod, 207
fldiv0, 208
fxand, 197
fxarithmetic-shift, 201
fxarithmetic-shift-left, 201
fxarithmetic-shift-right, 201
fxbit-count, 198
fxbit-field, 200
fxbit-set?, 199
fxcopy-bit, 200
fxcopy-bit-field, 200
fxdiv, 196
fxdiv-and-mod, 196
fxdiv0, 196
fxdiv0-and-mod0, 196
fxeven?, 194
fxfirst-bit-set, 199
fxif, 198
fxior, 197
fxlength, 198
fxmax, 195
fxmin, 195
fxmod, 196
fxmod0, 196
fxnegative?, 194
fxnot, 197
fxodd?, 194
fxpositive?, 194
fxreverse-bit-field, 202
fxrotate-bit-field, 201
fxxor, 197
fxzero?, 194
garbage collector, 3
gcd, 179
generate-temporaries, 310
generative, 324
get-bytevector-all, 275
get-bytevector-n, 274
get-bytevector-n!, 274
get-bytevector-some, 275
get-char, 275
get-datum, 278
get-line, 277
get-string-all, 277
get-string-n, 276
get-string-n!, 276
get-u8, 274
getq, 54
goodbye, 41
greatest-fixnum, 193
guard, 361
hare and tortoise, 56, 66
hashtable-clear!, 249
hashtable-contains?, 246
hashtable-copy, 248
hashtable-delete!, 248
hashtable-entries, 250
hashtable-equivalence-function, 245
hashtable-hash-function, 245
hashtable-keys, 249
hashtable-mutable?, 245
hashtable-ref, 246
hashtable-set!, 246
hashtable-size, 248
hashtable-update!, 247
hashtable?, 155
hashtables, 243
i/o-decoding-error?, 375
i/o-encoding-error-char, 376
i/o-encoding-error?, 376
i/o-error-filename, 373
i/o-error-port, 375
i/o-error-position, 372
i/o-error?, 371
i/o-file-already-exists-error?, 374
i/o-file-does-not-exist-error?, 374
i/o-file-is-read-only-error?, 374
i/o-file-protection-error?, 373
i/o-filename-error?, 373
i/o-invalid-position-error?, 372
i/o-port-error?, 375
i/o-read-error?, 372
i/o-write-error?, 372
identifier-syntax, 291, 297, 307, 316, 317
identifier?, 301
identifiers, 6
if, 35, 36, 39, 51, 59, 109
imag-part, 182
immutability of exports, 349
immutable, 331
implementation-restriction-violation?, 369
implicit begin, 109
import, 345
import level, 345
import spec, 345, 346
improper list, 19, 155
include, 309
indirect exports, 349
inexact, 180
inexact->exact, 181
inexact?, 167, 170
infinite?, 174
inheritance, 412
inheritance in records, 325
input port, 257
input-port?, 270
integer->char, 215
integer-divide, 79
integer-valued?, 153
integer?, 151, 167
The Scheme Programming Language, 4th Edition

Index

log, 184
lookahead-char, 275
lookahead-u8, 274
loop, 308
looping, 5
macros, 291
magnitude, 178, 183
make-assertion-violation, 366
make-bytevector, 228
make-counter, 50, 54
make-custom-binary-input-port, 267
make-custom-binary-input/output-port, 267
make-custom-binary-output-port, 267
make-custom-textual-input-port, 268
make-custom-textual-input/output-port, 268
make-custom-textual-output-port, 268
make-enumeration, 251
make-eq-hashtable, 243
make-eqv-hashtable, 244
make-error, 367
make-hashtable, 244
make-i/o-decoding-error, 375
make-i/o-encoding-error, 376
make-i/o-error, 371
make-i/o-file-already-exists-error, 374
make-i/o-file-does-not-exist-error, 374
make-i/o-file-is-read-only-error, 374
make-i/o-file-protection-error, 373
make-i/o-filename-error, 373
make-i/o-invalid-position-error, 372
make-i/o-port-error, 375
make-i/o-read-error, 372
make-i/o-write-error, 372
make-implementation-restriction-violation, 369
make-irritants-condition, 368
make-lexical-violation, 370
make-list, 46, 94
make-message-condition, 368
make-no-infinities-violation, 376
make-no-nans-violation, 377
make-non-continuable-violation, 369
make-polar, 183
make-promise, 129
make-queue, 54
make-record-constructor-descriptor, 332
make-record-type-descriptor, 323, 331
make-record-type-descriptor, 331
make-rectangular, 182
make-serious-condition, 366
make-stack, 52, 55
make-string, 218
make-syntax-violation, 370
make-transcoder, 259
make-undefined-violation, 371
make-variable-transformer, 291, 298, 306
make-vector, 224
make-violation, 366
make-warning, 367
make-who-condition, 369
map, 45, 47, 117, 392
map!, 46
mapping, 45, 117, 118, 121, 122
matrix multiplication, 381
max, 178
member, 161
memp, 163
memq, 161
memv, 43, 161
merge, 387
message-condition?, 368
messages, 52, 408
meta-circular interpreter, 404
method, 317
min, 178
mod, 175
mod0, 176
modulo, 175
mul, 382
multiple values, 9
multiprocessing, 421
mutable, 331
mutually recursive procedures, 66, 97
named let, 67, 71, 114
naming conventions, 8
nan?, 174
native-endianness, 228
native-eol-style, 260
native-transcoder, 259
negative?, 173
nested engines, 429
nested let expressions, 96
newline, 285
no-infinities-violation?, 376
no-nans-violation?, 377
non-continuable-violation?, 369
nondeterministic computations, 421, 424
nongenerative, 331
nongenerative, 324
nonlocal exits, 123, 124
not, 36, 110
null-environment, 137
null?, 37, 151
number syntax, 459
number->string, 191
number?, 38, 151
numbers, 16, 167
numerator, 181
object identity, 144
object->string, 267
object-oriented programming, 317, 408
objects, 3
occur free, 28, 30
octet, 257
odd?, 47, 66, 81, 174
only import set, 346
opaque, 331
opaque record type, 330, 336
open-bytevector-input-port, 264
open-bytevector-output-port, 265
open-file-input-port, 262
open-file-input/output-port, 263
open-file-output-port, 262
open-input-file, 280
open-output-file, 281
open-string-input-port, 265
open-string-output-port, 266
operating system, 423, 429
operations on objects, 141
operator precedence, 16
optional arguments, 93
or, 36, 63, 110
order of evaluation, 22, 107
output port, 257
output-port-buffer-mode, 273
output-port?, 270
pair?, 38, 151
pairs, 19, 155
parent, 331
parent type, 325
parent-rtd, 331
partition, 164
pattern variable, 294
pattern variables, 61, 299
patterns, 294
peek-char, 284
Petite Chez Scheme, ix
pointers, 4
por (parallel-or), 424
port, 257
port-eof?, 278
port-has-port-position?, 271
port-has-set-port-position!?, 272
port-position, 271
port-transcoder, 271
port?, 270
positive?, 173
predicates, 8, 37, 143
prefix import set, 346
prefix notation, 15, 16
primitive procedures, 4
printf, 401
procedure application, 16, 17, 21, 27, 107
procedure definition, 5, 31, 100
procedure?, 155
procedures, 26, 91, 92
product, 74, 80
proper list, 19, 56, 155
protocol, 331
protocol for records, 326, 332
put-bytevector, 279
put-char, 279
put-datum, 279, 397
put-string, 279
put-u8, 278
putq!, 54
quadratic-formula, 48
quasiquote ('), 142
quasisyntax (#*), 305
question mark (?), 8, 37
queue, 53
quote ('), 17, 22, 59, 141
quotient, 175
raise, 357
raise-continuable, 357
rational numbers, 167
rational-valued?, 153
rational?, 151, 167
rationalize, 181
rcd, 332
read, 284
read-char, 284
real numbers, 167
real->flonum, 211
real-part, 182
real-valued?, 153
real?, 151, 167
rec, 311
reciprocal, 15, 37, 39, 80
record generativity, 324
record inheritance, 325
record uid, 325
record-accessor, 334
record-constructor, 333
record-constructor descriptor, 332
record-constructor-descriptor, 333
record-field-mutable?, 338
record-mutator, 334
record-predicate, 333
record-rtd, 338
record-type descriptor, 331
record-type-descriptor, 333
record-type-descriptor?, 332
record-type-field-names, 337
record-type-generative?, 337
record-type-name, 336
record-type-opaque?, 337
Index

record-type-parent, 336
record-type-sealed?, 337
record-type-uid, 336
record?, 338
records, 323
recursion, 5, 41, 65, 114
recursion step, 41
recursive procedure, 41
remainder, 175
remove, 163
remp, 163
remq, 163
remv, 44, 163
rename import set, 346
retry, 75, 80
reverse, 161
Revised Reports, ix, 3
round, 178
round-robin, 423
rtd, 331
s (short), 169
Scheme standard, ix
scheme-report-environment, 137
scope, 25
sealed, 331
sealed record type, 330
segment-length, 132
segment-slope, 132
semicolon ( ; ), 7, 455
sequence, 313
sequencing, 108
serious-condition?, 366
set!, 47, 59, 102
set-car!, 157
set-cdr!, 56, 157
set-of, 389
set-port-position!, 272
sets, 389
shadowing, 4, 25, 31
shhh, 50
shorter, 41, 47
shorter?, 47
side effects, 8, 108
simple condition, 362
simple-conditions, 363
sin, 185
sint-list->bytevector, 239
sort, 387
split, 133
sqrt, 183
square, 14
stack objects, 52
standard-error-port, 264
standard-input-port, 264
standard-output-port, 264
streams, 128
string, 218
string syntax, 458
string->bytevector, 287
string->list, 222
string->number, 191
string->symbol, 242
string->utf16, 287
string->utf32, 287
string->utf8, 287
string-append, 219
string-ci-hash, 245
string-ci<=?, 217
string-ci<?, 217
string-ci=?, 217
string-ci>=?, 217
string-ci>?, 217
string-copy, 219
string-downcase, 221
string-fill!, 220
string-foldcase, 221
string-for-each, 122
string-hash, 245
string-length, 218
string-normalize-nfc, 222
string-normalize-nfd, 222
string-normalize-nfkc, 222
string-normalize-nfkd, 222
string-ref, 218
string-set!, 219
string-titlecase, 221
string-upcase, 221
string<=?, 216
string<?, 216
string>=?, 216
string>?, 216
string?, 38, 154
strings, 14, 216
structured forms, 6
structures, 318
substring, 95, 220
sum, 65
symbol syntax, 458
symbol table, 241
symbol->string, 242
symbol-hash, 245
symbol=?, 242
symbol?, 38, 154
symbols, 18, 241
syntactic extensions, 5, 22, 59, 60, 291
syntactic forms, 18, 59, 291
syntax, 291

392
syntax (#'), 300
syntax object, 298
syntax violation, 9
syntax->datum, 308
syntax-case, 291, 299
syntax-rules, 291, 294, 300, 389
syntax-violation, 359
syntax-violation-form, 370
syntax-violation-subform, 370
syntax-violation?, 370
tail call, 5, 68
tail recursion, 5, 68
tan, 185
tconc, 53
tell, 50
templates, 295
textual port, 257
textual-port?, 270
threads, 421
thunk, 51, 124
ticks, see engines
timed preemption, 421
timer interrupts, 425	tokens, 455
top-level definitions, 30, 101
top-level programs, 343
trace, 42
tracing, 42
transcoded-port, 271
transcoder, 257
transcoder-c codec, 259
transcoder-eol-style, 259
transcoder-error-handling-mode, 259
transformer, 61
tree-copy, 44
true, 7, 36
truncate, 177
type predicates, 38
u8-list->bytevector, 232
uint-list->bytevector, 239
undefined-violation?, 371
underscore (_), 61, 296, 315
underscore (___), 294
unification, 417
unless, 64, 112
unquote (, @), 142
unquote-splicing (, @), 142
unspecified, 9
unsyntax (#, @), 305
unsyntax-splicing (#, @), 305
unwind-protect (in Lisp), 124
utf-16, 257
utf-16-c codec, 259
utf-8, 257  
utf-8-codec, 259  
utf16->string, 288  
utf32->string, 288  
utf8->string, 287  
values, 130, 131  
variable binding, 23, 91  
variable reference, 91  
variables, 4, 18, 23, 30, 47  
vector, 224  
vector syntax, 461  
vector->list, 225  
vector-fill!, 225  
vector-for-each, 122  
vector-length, 224  
vector-map, 121  
vector-ref, 224  
vector-set!, 225  
vector-sort, 226  
vector-sort!, 226  
vector?, 154  
vectors, 55, 223, 383  
violation?, 366  
warning?, 367  
when, 64, 112  
whitespace, 455  
whitespace characters, 7  
who-condition?, 369  
winders, see dynamic-wind  
with-exception-handler, 360  
with-input-from-file, 283  
with-output-to-file, 283  
with-syntax, 304  
write, 284, 397  
write-char, 285  
x++, 316  
zero?, 173