Revised^6 Report on the Algorithmic Language Scheme

Michael Sperber
R. Kent Dybvig, Matthew Flatt, Anton van Straaten
(*Editors*)
Richard Kelsey, William Clinger, Jonathan Rees
(*Editors, Revised*^{5} *Report on the Algorithmic Language Scheme*)
Robert Bruce Findler, Jacob Matthews
(*Authors, formal semantics*)

~~**26 September 2007**~~

Unofficial version incorporating errata

`https://standards.scheme.org/`

21 July 2019

SUMMARY

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

This report is accompanied by a report describing standard libraries [24]; references to this document are identified by designations such as "library section" or "library chapter". It is also accompanied by a report containing non-normative appendices [22]. A fourth report gives some historical background and rationales for many aspects of the language and its libraries [23].

The individuals listed above are not the sole authors of the text of the report. Over the years, the following individuals were involved in discussions contributing to the design of the Scheme language, and were listed as authors of prior reports:

Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, R. Kent Dybvig, Daniel Friedman, Robert Halstead, Chris Hanson, Christopher Haynes, Eugene Kohlbecker, Don Oxley, Kent Pitman, Jonathan Rees, Guillermo Rozas, Guy L. Steele Jr., Gerald Jay Sussman, and Mitchell Wand.

In order to highlight recent contributions, they are not listed as authors of this version of the report. However, their contribution and service is gratefully acknowledged.

CONTENTS

INTRODUCTION

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

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

Guiding principles

To help guide the standardization effort, the editors have adopted a set of principles, presented below. Like the Scheme language defined in *Revised*^{5} *Report on the Algorithmic Language Scheme* [14], the language described in this report is intended to:

- allow programmers to read each other's code, and allow development of portable programs that can be executed in any conforming implementation of Scheme;
- derive its power from simplicity, a small number of generally useful core syntactic forms and procedures, and no unnecessary restrictions on how they are composed;
- allow programs to define new procedures and new hygienic syntactic forms;
- support the representation of program source code as data;
- make procedure calls powerful enough to express any form of sequential control, and allow programs to perform non-local control operations without the use of global program transformations;
- allow interesting, purely functional programs to run indefinitely without terminating or running out of memory on finite-memory machines;
- allow educators to use the language to teach programming effectively, at various levels and with a variety of pedagogical approaches; and
- allow researchers to use the language to explore the design, implementation, and semantics of programming languages.

In addition, this report is intended to:

- allow programmers to create and distribute substantial programs and libraries, e.g., implementations of Scheme Requests for Implementation, that run without modification in a variety of Scheme implementations;
- support procedural, syntactic, and data abstraction more fully by allowing programs to define hygiene-bending and hygiene-breaking syntactic abstractions and new unique datatypes along with procedures and hygienic macros in any scope;
- allow programmers to rely on a level of automatic run-time type and bounds checking sufficient to ensure type safety; and
- allow implementations to generate efficient code, without requiring programmers to use implementation-specific operators or declarations.

While it was possible to write portable programs in Scheme as described in *Revised*^{5} *Report on the Algorithmic Language Scheme*, and indeed portable Scheme programs were written prior to this report, many Scheme programs were not, primarily because of the lack of substantial standardized libraries and the proliferation of implementation-specific language additions.

In general, Scheme should include building blocks that allow a wide variety of libraries to be written, include commonly used user-level features to enhance portability and readability of library and application code, and exclude features that are less commonly used and easily implemented in separate libraries.

The language described in this report is intended to also be backward compatible with programs written in Scheme as described in *Revised⁵ Report on the Algorithmic Language Scheme* to the extent possible without compromising the above principles and future viability of the language. With respect to future viability, the editors have operated under the assumption that many more Scheme programs will be written in the future than exist in the present, so the future programs are those with which we should be most concerned.

Acknowledgements

Many people contributed significant help to this revision of the report. Specifically, we thank Aziz Ghuloum and André van Tonder for contributing reference implementations of the library system. We thank Alan Bawden, John Cowan, Sebastian Egner, Aubrey Jaffer, Shiro Kawai, Bradley Lucier, and André van Tonder for contributing insights on language design. Marc Feeley, Martin Gasbichler, Aubrey Jaffer, Lars T Hansen, Richard Kelsey, Olin Shivers, and André van Tonder wrote SRFIs that served as direct input to the report. Casey Klein found and fixed several bugs in the formal semantics. Marcus Crestani, David Frese, Aziz Ghuloum, Arthur A. Gleckler, Eric Knauel, Jonathan Rees, and André van Tonder thoroughly proofread early versions of the report.

We would also like to thank the following people for their help in creating this report: Lauri Alanko, Eli Barzilay, Alan Bawden, Brian C. Barnes, Per Bothner, Trent Buck, Thomas Bushnell, Taylor Campbell, Ludovic Courtès, Pascal Costanza, John Cowan, Ray Dillinger, Jed Davis, J.A. "Biep" Durieux, Carl Eastlund, Sebastian Egner, Tom Emerson, Marc Feeley, Matthias Felleisen, Andy Freeman, Ken Friedenbach, Martin Gasbichler, Arthur A. Gleckler, Aziz Ghuloum, Dave Gurnell, Lars T Hansen, Ben Harris, Sven Hartrumpf, Dave Herman, Nils M. Holm, Stanislav Ievlev, James Jackson, Aubrey Jaffer, Shiro Kawai, Alexander Kjeldaas, Eric Knauel, Michael Lenaghan, Felix Klock, Donovan Kolbly, Marcin Kowalczyk, Thomas Lord, Bradley Lucier, Paulo J. Matos, Dan Muresan, Ryan Newton, Jason Orendorff, Erich Rast, Jeff Read, Jonathan Rees, Jorgen Schäfer, Paul Schlie, Manuel Serrano, Olin Shivers, Jonathan Shapiro, Jens Axel Søgaard, Jay Sulzberger, Pinku Surana, Mikael Tillenius, Sam Tobin-Hochstadt, David Van Horn, André van Tonder, Reinder Verlinde, Alan Watson, Andrew Wilcox, Jon Wilson, Lynn Winebarger, Keith Wright, and Chongkai Zhu.

We would like to thank the following people for their help in creating the previous revisions of this report: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Marc Feeley, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu, and Ozan Yigit.

We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual* [26]. We gladly acknowledge the influence of manuals for MIT Scheme [20], T [21], Scheme 84 [12], Common Lisp [25], Chez Scheme [8], PLT Scheme [11], and Algol 60 [1].

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

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

DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

This chapter gives an overview of Scheme's semantics. The purpose of this overview is to explain enough about the basic concepts of the language to facilitate understanding of the subsequent chapters of the report, which are organized as a reference manual. Consequently, this overview is not a complete introduction to the language, nor is it precise in all respects or normative in any way.

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

Scheme has latent as opposed to manifest types [28]. Types are associated with objects (also called values) rather than with variables. (Some authors refer to languages with latent types as untyped, weakly typed or dynamically typed languages.) Other languages with latent types are Python, Ruby, Smalltalk, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, C, C#, Java, Haskell, and ML.

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

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

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

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

In Scheme, the argument expressions of a procedure call are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. C, C#, Common Lisp, Python, Ruby, and Smalltalk are other languages that always evaluate argument expressions before invoking a procedure. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

Scheme's model of arithmetic provides a rich set of numerical types and operations on them. Furthermore, it distinguishes *exact* and *inexact* number objects: Essentially, an exact number object corresponds to a number exactly, and an inexact number object is the result of a computation that involved rounding or other errors.

1.1. Basic types

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

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

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

Numbers Scheme supports a rich variety of numerical data types, including objects representing integers of arbitrary precision, rational numbers, complex numbers, and inexact numbers of various kinds. Chapter 3 gives an overview of the structure of Scheme's numerical tower.

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

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

Symbols A symbol is an object representing a string, the symbol's *name*. Unlike strings, two symbols whose names are spelled the same way are never distinguishable. Symbols are useful for many applications; for instance, they may be used the way enumerated values are used in other languages.

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

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

Procedures Procedures are values in Scheme.

1.2. Expressions

The most important elements of Scheme code are *expressions*. Expressions can be *evaluated*, producing a *value*. (Actually, any number of values—see section 5.8.) The most fundamental expressions are literal expressions:

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

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

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

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

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

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

As in many other languages, whitespace (including line endings) is not significant when it separates subexpressions of an expression, and can be used to indicate structure.

1.3. Variables and binding

Scheme allows identifiers to stand for locations containing values. These identifiers are called variables. In many cases, specifically when the location's value is never modified after its creation, it is useful to think of the variable as standing for the value directly.

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

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

1.4. Definitions

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

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

(These are actually "top-level" in the body of a top-level program or library; see section 1.12 below.)

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

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

```
(define x 23)
(define y 42)
(let ((y 43))
```

```
  (+ x y))                    ⟹ 66

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

1.5. Forms

While definitions are not expressions, compound expressions and definitions exhibit similar syntactic structure:

```
(define x 23)
(* x 2)
```

While the first line contains a definition, and the second an expression, this distinction depends on the bindings for `define` and `*`. At the purely syntactical level, both are *forms*, and *form* is the general name for a syntactic part of a Scheme program. In particular, `23` is a *subform* of the form `(define x 23)`.

1.6. Procedures

Definitions can also be used to define procedures:

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

(f 23)                        ⟹ 65
```

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

As procedures are objects, they can be passed to other procedures:

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

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

(g f 23)                      ⟹ 65
```

In this example, the body of `g` is evaluated with `p` bound to `f` and `x` bound to 23, which is equivalent to `(f 23)`, which evaluates to 65.

In fact, many predefined operations of Scheme are provided not by syntax, but by variables whose values are procedures. The `+` operation, for example, which receives special syntactic treatment in many other languages, is just a regular identifier in Scheme, bound to a procedure that adds number objects. The same holds for `*` and many others:

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

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

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

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

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

1.7. Procedure calls and syntactic keywords

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

The rules for the `lambda` keyword specify that the first subform is a list of parameters, and the remaining subforms are the body of the procedure. In `let` expressions, the first subform is a list of binding specifications, and the remaining subforms constitute a body of expressions.

Procedure calls can generally be distinguished from these *special forms* by looking for a syntactic keyword in the first position of an form: if the first position does not contain a syntactic keyword, the expression is a procedure call. (So-called *identifier macros* allow creating other kinds of special forms, but are comparatively rare.) The set of syntactic keywords of Scheme is fairly small, which usually makes this task fairly simple. It is possible, however, to create new bindings for syntactic keywords; see section 1.9 below.

1.8. Assignment

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

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

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

1.9. Derived forms and macros

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

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

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

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

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

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

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

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

(def f (x)
  (+ x 42))
```

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

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

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

1.10. Syntactic data and datum values

A subset of the Scheme objects is called *datum values*. These include booleans, number objects, characters, symbols, and strings as well as lists and vectors whose elements are data. Each datum value may be represented in textual form as a *syntactic datum*, which can be written out and read back in without loss of information. A datum value may be represented by several different syntactic data. Moreover, each datum value can be trivially translated to a literal expression in a program by prepending a `'` to a corresponding syntactic datum:

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

The `'` shown in the previous examples is not needed for representations of number objects or booleans. The syntactic datum `foo` represents a symbol with name "foo", and `'foo` is a literal expression with that symbol as its value. `(1 2 3)` is a syntactic datum that represents a list with elements 1, 2, and 3, and `'(1 2 3)` is a literal expression with this list as its value. Likewise, `#(1 2 3)` is a syntactic datum that represents a vector with elements 1, 2 and 3, and `'#(1 2 3)` is the corresponding literal.

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

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

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

1.11. Continuations

Whenever a Scheme expression is evaluated there is a *continuation* wanting the result of the expression. The continuation represents an entire (default) future for the computation. For example, informally the continuation of `3` in the expression

```
(+ 1 3)
```

adds 1 to it. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. The `call-with-current-continuation` procedure (see section 11.15) allows Scheme programmers to do that by creating a procedure that reinstates the current continuation. The `call-with-current-continuation` procedure accepts a procedure, calls it immediately with an argument that is an *escape procedure*. This escape procedure can then be called with an argument that becomes the result of the call to `call-with-current-continuation`. That is, the escape procedure abandons its own continuation, and reinstates the continuation of the call to `call-with-current-continuation`.

In the following example, an escape procedure representing the continuation that adds 1 to its argument is bound to `escape`, and then called with 3 as an argument. The continuation of the call to `escape` is abandoned, and instead the 3 is passed to the continuation that adds 1:

```
(+ 1 (call-with-current-continuation
       (lambda (escape)
         (+ 2 (escape 3)))))
          ⟹ 4
```

An escape procedure has unlimited extent: It can be called after the continuation it captured has been invoked, and it can be called multiple times. This makes `call-with-current-continuation` significantly more powerful than typical non-local control constructs such as exceptions in other languages.

1.12. Libraries

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

The following library called `(hello)` exports a definition called `hello-world`, and imports the base library (see chapter 11) and the simple I/O library (see library section 8.3). The `hello-world` export is a procedure that displays `Hello World` on a separate line:

```
(library (hello)
  (export hello-world)
  (import (rnrs base)
          (rnrs io simple))
  (define (hello-world)
    (display "Hello World")
    (newline)))
```

1.13. Top-level programs

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

The following top-level program obtains the first argument from the command line via the `command-line` procedure from the `(rnrs programs (6))` library (see library chapter 10). It then opens the file using `open-file-input-port` (see library section 8.2), yielding a *port*, i.e. a connection to the file as a data source, and calls the `get-bytevector-all` procedure to obtain the contents of the file as binary data. It then uses `put-bytevector` to output the contents of the file to standard output:

```
#!r6rs
(import (rnrs base)
        (rnrs io ports)
        (rnrs programs))
(let ((p (standard-output-port)))
  (put-bytevector p
                  (call-with-port
                      (open-file-input-port
                        (cadr (command-line)))
                    get-bytevector-all))
  (close-port p))
```

2. Requirement levels

The key words "must", "must not", "should", "should not", "recommended", "may", and "optional" in this report are to be interpreted as described in RFC 2119 [3]. Specifically:

must This word means that a statement is an absolute requirement of the specification.

must not This phrase means that a statement is an absolute prohibition of the specification.

should This word, or the adjective "recommended", means that valid reasons may exist in particular circumstances to ignore a statement, but that the implications must be understood and weighed before choosing a different course.

should not This phrase, or the phrase "not recommended", means that valid reasons may exist in particular circumstances when the behavior of a statement is acceptable, but that the implications should be understood and weighed before choosing the course described by the statement.

may This word, or the adjective "optional", means that an item is truly optional.

In particular, this report occasionally uses "should" to designate circumstances that are outside the specification of this report, but cannot be practically detected by an implementation; see section 5.4. In such circumstances, a particular implementation may allow the programmer to ignore the recommendation of the report and even exhibit reasonable behavior. However, as the report does not specify the behavior, these programs may be unportable, that is, their execution might produce different results on different implementations.

Moreover, this report occasionally uses the phrase "not required" to note the absence of an absolute requirement.

3. Numbers

This chapter describes Scheme's model for numbers. It is important to distinguish between the mathematical numbers, the Scheme objects that attempt to model them, the machine representations used to implement the numbers, and notations used to write numbers. In this report, the term *number* refers to a mathematical number, and the term *number object* refers to a Scheme object representing a number. This report uses the types *complex*, *real*, *rational*, and *integer* to refer to both mathematical numbers and number objects. The *fixnum* and *flonum* types refer to special subsets of the number objects, as determined by common machine representations, as explained below.

3.1. Numerical tower

Numbers may be arranged into a tower of subsets in which each level is a subset of the level above it:

number
complex
real
rational
integer

For example, 5 is an integer. Therefore 5 is also a rational, a real, and a complex. The same is true of the number objects that model 5.

Number objects are organized as a corresponding tower of subtypes defined by the predicates `number?`, `complex?`, `real?`, `rational?`, and `integer?`; see section 11.7.4. Integer number objects are also called *integer objects*.

There is no simple relationship between the subset that contains a number and its representation inside a computer. For example, the integer 5 may have several representations. Scheme's numerical operations treat number objects as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use many different representations for numbers, this should not be apparent to a casual programmer writing simple programs.

3.2. Exactness

It is useful to distinguish between number objects that are known to correspond to a number exactly, and those number objects whose computation involved rounding or other errors. For example, index operations into data structures may need to know the index exactly, as may some operations on polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of numbers known only inexactly where exact numbers are required, Scheme explicitly distinguishes *exact* from *inexact* number objects. This distinction is orthogonal to the dimension of type.

A number object is exact if it is the value of an exact numerical literal or was derived from exact number objects using only exact operations. Exact number objects correspond to mathematical numbers in the obvious way.

Conversely, a number object is inexact if it is the value of an inexact numerical literal, or was derived from inexact number objects, or was derived using inexact operations. Thus inexactness is contagious.

Exact arithmetic is reliable in the following sense: If exact number objects are passed to any of the arithmetic procedures described in section 11.7.1, and an exact number object is returned, then the result is mathematically correct. This is generally not true of computations involving inexact number objects because approximate methods such as floating-point arithmetic may be used, but it is the duty of each implementation to make the result as close as practical to the mathematically ideal result.

3.3. Fixnums and flonums

A *fixnum* is an exact integer object that lies within a certain implementation-dependent subrange of the exact integer objects. (Library section 11.2 describes a library for computing with fixnums.) Likewise, every implementation must designate a subset of its inexact real number objects as *flonum*s, and to convert certain external representations into flonums. (Library section 11.3 describes a library for computing with flonums.) Note that this does not imply that an implementation must use floating-point representations.

3.4. Implementation requirements

Implementations of Scheme must support number objects for the entire tower of subtypes given in section 3.1. Moreover, implementations must support exact integer objects and exact rational number objects of practically unlimited size and precision, and to implement certain procedures (listed in 11.7.1) so they always return exact results when given exact arguments. ("Practically unlimited" means that the size and precision of these numbers should only be limited by the size of the available memory.)

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

An implementation may use floating point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that the IEEE floating-point standards be followed by implementations that use floating-point representations, and that implementations using other representations should match or exceed the precision achievable using these floating-point standards [13].

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

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

3.5. Infinities and NaNs

Some Scheme implementations, specifically those that follow the IEEE floating-point standards, distinguish special number objects called *positive infinity*, *negative infinity*, and *NaN*.

Positive infinity is regarded as an inexact real (but not rational) number object that represents an indeterminate number greater than the numbers represented by all rational number objects. Negative infinity is regarded as an inexact real (but not rational) number object that represents an indeterminate number less than the numbers represented by all rational numbers.

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

3.6. Distinguished -0.0

Some Scheme implementations, specifically those that follow the IEEE floating-point standards, distinguish between number objects for 0.0 and -0.0, i.e., positive and negative inexact zero. This report will sometimes specify the behavior of certain arithmetic operations on these number objects. These specifications are marked with "if -0.0 is distinguished" or "implementations that distinguish -0.0".

4. Lexical syntax and datum syntax

The syntax of Scheme code is organized in three levels:

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

Syntactic data (also called *external representations*) double as a notation for objects, and Scheme's `(rnrs io ports (6))` library (library section 8.2) provides the `get-datum` and `put-datum` procedures for reading and writing syntactic data, converting between their textual representation and the corresponding objects. Each syntactic datum represents a corresponding *datum value*. A syntactic datum can be used in a program to obtain the corresponding datum value using `quote` (see section 11.4.1).

Scheme source code consists of syntactic data and (non-significant) comments. Syntactic data in Scheme source code are called *forms*. (A form nested inside another form is called a *subform*.) Consequently, Scheme's syntax has the property that any sequence of characters that is a form is also a syntactic datum representing some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to be a representation of objects or the text of a program. It is also a source of power, since it facilitates writing programs such as interpreters or compilers that treat programs as objects (or vice versa).

A datum value may have several different external representations. For example, both "`#e28.000`" and "`#x1c`" are syntactic data representing the exact integer object 28, and the syntactic data "`(8 13)`", "`( 08 13 )`", "`(8 . (13 .`

`()))`” all represent a list containing the exact integer objects 8 and 13. Syntactic data that represent equal objects (in the sense of `equal?`; see section 11.5) are always equivalent as forms of a program.

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

An implementation must not extend the lexical or datum syntax in any way, with one exception: it need not treat the syntax `#!`⟨identifier⟩, for any ⟨identifier⟩ (see section 4.2.4) that is not `r6rs`, as a syntax violation, and it may use specific `#!`-prefixed identifiers as flags indicating that subsequent input contains extensions to the standard lexical or datum syntax. The syntax `#!r6rs` may be used to signify that the input afterward is written with the lexical syntax and datum syntax described by this report. `#!r6rs` is otherwise treated as a comment; see section 4.2.3.

4.1. Notation

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

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

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

Some non-terminal names refer to the Unicode scalar values of the same name: ⟨character tabulation⟩ (U+0009), ⟨linefeed⟩ (U+000A), ⟨line tabulation⟩ (U+000B), ⟨form feed⟩ (U+000C), ⟨carriage return⟩ (U+000D), ⟨space⟩ (U+0020), ⟨next line⟩ (U+0085), ⟨line separator⟩ (U+2028), and ⟨paragraph separator⟩ (U+2029).

4.2. Lexical syntax

The lexical syntax determines how a character sequence is split into a sequence of lexemes, omitting non-significant portions such as comments and whitespace. The character sequence is assumed to be text according to the Unicode standard [27]. Some of the lexemes, such as identifiers, representations of number objects, strings etc., of the lexical syntax are syntactic data in the datum syntax, and thus represent objects. Besides the formal account of the syntax, this section also describes what datum values are represented by these syntactic data.

The lexical syntax, in the description of comments, contains a forward reference to ⟨datum⟩, which is described as part of the datum syntax. Being comments, however, these ⟨datum⟩s do not play a significant role in the syntax.

Case is significant except in representations of booleans, number objects, and in hexadecimal numbers specifying Unicode scalar values. For example, `#x1A` and `#X1a` are equivalent. The identifier `Foo` is, however, distinct from the identifier `FOO`.

4.2.1. Formal account

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

⟨Identifier⟩s, `.`, ⟨number⟩s, ⟨character⟩s, and ⟨boolean⟩s, must be terminated by a ⟨delimiter⟩ or by the end of the input.

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

```
⟨lexeme⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
     | ⟨character⟩ | ⟨string⟩
     | ( | ) | [ | ] | #( | #vu8( | ' | ` | , | ,@ | .
     | #' | #` | #, | #,@
⟨delimiter⟩ ⟶ ( | ) | [ | ] | " | ; | #
     | ⟨whitespace⟩
⟨whitespace⟩ ⟶ ⟨character tabulation⟩
     | ⟨linefeed⟩ | ⟨line tabulation⟩ | ⟨form feed⟩
     | ⟨carriage return⟩ | ⟨next line⟩
     | ⟨any character whose category is Zs, Zl, or Zp⟩
⟨line ending⟩ ⟶ ⟨linefeed⟩ | ⟨carriage return⟩
     | ⟨carriage return⟩ ⟨linefeed⟩ | ⟨next line⟩
     | ⟨carriage return⟩ ⟨next line⟩ | ⟨line separator⟩
⟨comment⟩ ⟶ ; ⟨all subsequent characters up to a
                 ⟨line ending⟩ or ⟨paragraph separator⟩⟩
     | ⟨nested comment⟩
     | #; ⟨interlexeme space⟩ ⟨datum⟩
     | #!r6rs
⟨nested comment⟩ ⟶ #| ⟨comment text⟩
                      ⟨comment cont⟩* |#
⟨comment text⟩ ⟶ ⟨character sequence not containing
                    #| or |#⟩
⟨comment cont⟩ ⟶ ⟨nested comment⟩ ⟨comment text⟩
⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩
⟨interlexeme space⟩ ⟶ ⟨atmosphere⟩*

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

⟨special initial⟩ ⟶ ! | $ | % | & | * | / | : | < | =
| > | ? | ^ | _ | ~
⟨subsequent⟩ ⟶ ⟨initial⟩ | ⟨digit⟩
| ⟨any character whose category is Nd, Mc, or Me⟩
| ⟨special subsequent⟩
⟨digit⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨hex digit⟩ ⟶ ⟨digit⟩
| a | A | b | B | c | C | d | D | e | E | f | F
⟨special subsequent⟩ ⟶ + | - | . | @
⟨inline hex escape⟩ ⟶ \x⟨hex scalar value⟩;
⟨hex scalar value⟩ ⟶ ⟨hex digit⟩$^{+}$
⟨peculiar identifier⟩ ⟶ + | - | ... | -> ⟨subsequent⟩*
⟨boolean⟩ ⟶ #t | #T | #f | #F
⟨character⟩ ⟶ #\⟨any character⟩
| #\⟨character name⟩
| #\x⟨hex scalar value⟩
⟨character name⟩ ⟶ nul | alarm | backspace | tab
| linefeed | newline | vtab | page | return
| esc | space | delete
⟨string⟩ ⟶ " ⟨string element⟩* "
⟨string element⟩ ⟶ ⟨any character other than " or \⟩
| \a | \b | \t | \n | \v | \f | \r
| \" | \\
| \⟨intraline whitespace⟩*⟨line ending⟩
⟨intraline whitespace⟩*
| ⟨inline hex escape⟩
⟨intraline whitespace⟩ ⟶ ⟨character tabulation⟩
| ⟨any character whose category is Zs⟩

A ⟨hex scalar value⟩ represents a Unicode scalar value between 0 and #x10FFFF, excluding the range [#*x*D800, #*x*DFFF].

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

In the following rules, case is insignificant.

⟨number⟩ ⟶ ⟨num 2⟩ | ⟨num 8⟩
| ⟨num 10⟩ | ⟨num 16⟩
⟨num *R*⟩ ⟶ ⟨prefix *R*⟩ ⟨complex *R*⟩
⟨complex *R*⟩ ⟶ ⟨real *R*⟩ | ⟨real *R*⟩ @ ⟨real *R*⟩
| ⟨real *R*⟩ + ⟨ureal *R*⟩ i | ⟨real *R*⟩ - ⟨ureal *R*⟩ i
| ⟨real *R*⟩ + ⟨naninf⟩ i | ⟨real *R*⟩ - ⟨naninf⟩ i
| ⟨real *R*⟩ + i | ⟨real *R*⟩ - i
| + ⟨ureal *R*⟩ i | - ⟨ureal *R*⟩ i
| + ⟨naninf⟩ i | - ⟨naninf⟩ i
| + i | - i
⟨real *R*⟩ ⟶ ⟨sign⟩ ⟨ureal *R*⟩
| + ⟨naninf⟩ | - ⟨naninf⟩
⟨naninf⟩ ⟶ nan.0 | inf.0
⟨ureal *R*⟩ ⟶ ⟨uinteger *R*⟩
| ⟨uinteger *R*⟩ / ⟨uinteger *R*⟩
| ⟨decimal *R*⟩ ⟨mantissa width⟩
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
| . ⟨digit 10⟩$^{+}$ ⟨suffix⟩
| ⟨digit 10⟩$^{+}$. ⟨digit 10⟩* ⟨suffix⟩
⟨uinteger *R*⟩ ⟶ ⟨digit *R*⟩$^{+}$
⟨prefix *R*⟩ ⟶ ⟨radix *R*⟩ ⟨exactness⟩
| ⟨exactness⟩ ⟨radix *R*⟩

⟨suffix⟩ ⟶ ⟨empty⟩
| ⟨exponent marker⟩ ⟨sign⟩ ⟨digit 10⟩$^{+}$
⟨exponent marker⟩ ⟶ e | s | f
| d | l
⟨mantissa width⟩ ⟶ ⟨empty⟩
| | ⟨digit 10⟩$^{+}$
⟨sign⟩ ⟶ ⟨empty⟩ | + | -
⟨exactness⟩ ⟶ ⟨empty⟩
| #i | #e
⟨radix 2⟩ ⟶ #b
⟨radix 8⟩ ⟶ #o
⟨radix 10⟩ ⟶ ⟨empty⟩ | #d
⟨radix 16⟩ ⟶ #x
⟨digit 2⟩ ⟶ 0 | 1
⟨digit 8⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨hex digit⟩

4.2.2. Line endings

Line endings are significant in Scheme in single-line comments (see section 4.2.3) and within string literals. In Scheme source code, any of the line endings in ⟨line ending⟩ marks the end of a line. Moreover, the two-character line endings ⟨carriage return⟩ ⟨linefeed⟩ and ⟨carriage return⟩ ⟨next line⟩ each count as a single line ending.

In a string literal, a ⟨line ending⟩ not preceded by a \ stands for a linefeed character, which is the standard line-ending character of Scheme.

4.2.3. Whitespace and comments

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

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

A semicolon (`;`) indicates the start of a line comment. The comment continues to the end of the line on which the semicolon appears.

Another way to indicate a comment is to prefix a ⟨datum⟩ (cf. section 4.3.1) with `#;`, possibly with ⟨interlexeme space⟩ before the ⟨datum⟩. The comment consists of the comment prefix `#;` and the ⟨datum⟩ together. This notation is useful for "commenting out" sections of code.

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

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

The lexeme `#!r6rs`, which signifies that the program text that follows is written with the lexical and datum syntax described in this report, is also otherwise treated as a comment.

4.2.4. Identifiers

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

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

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

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

Moreover, all characters whose Unicode scalar values are greater than 127 and whose Unicode category is Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co can be used within identifiers. In addition, any character can be used within an identifier when specified via an ⟨inline hex escape⟩. For example, the identifier `H\x65;llo` is the same as the identifier `Hello`, and the identifier `\x3BB;` is the same as the identifier λ.

Any identifier may be used as a variable or as a syntactic keyword (see sections 5.2 and 9.2) in a Scheme program. Any identifier may also be used as a syntactic datum, in which case it represents a *symbol* (see section 11.10).

4.2.5. Booleans

The standard boolean objects for true and false have external representations `#t` and `#f`.

4.2.6. Characters

Characters are represented using the notation `#\`⟨character⟩ or `#\`⟨character name⟩ or `#\x`⟨hex scalar value⟩.

For example:

`#\a`	lower case letter a
`#\A`	upper case letter A
`#\(`	left parenthesis
`#\ `	space character
`#\nul`	U+0000
`#\alarm`	U+0007
`#\backspace`	U+0008
`#\tab`	U+0009
`#\linefeed`	U+000A
`#\newline`	U+000A
`#\vtab`	U+000B
`#\page`	U+000C
`#\return`	U+000D
`#\esc`	U+001B
`#\space`	U+0020
	preferred way to write a space
`#\delete`	U+007F
`#\xFF`	U+00FF
`#\x03BB`	U+03BB
`#\x00006587`	U+6587
`#\λ`	U+03BB
`#\x0001z`	`&lexical` *exception*
`#\λx`	`&lexical` *exception*
`#\alarmx`	`&lexical` *exception*
`#\alarm x`	U+0007
	followed by `x`
`#\Alarm`	`&lexical` *exception*
`#\alert`	`&lexical` *exception*
`#\xA`	U+000A
`#\xFF`	U+00FF
`#\xff`	U+00FF
`#\x ff`	U+0078
	followed by another datum, `ff`

`#\x(ff)`	U+0078 followed by another datum, a parenthesized `ff`
`#\(x)`	`&lexical` *exception*
`#\(x`	`&lexical` *exception*
`#\((x)`	U+0028 followed by another datum, parenthesized `x`
`#\x00110000`	`&lexical` *exception* out of range
`#\x000000001`	U+0001
`#\xD800`	`&lexical` *exception* in excluded range

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

Case is significant in `#\`⟨character⟩, and in `#\`⟨character name⟩, but not in the ⟨hex scalar value⟩ part of `#\x`⟨hex scalar value⟩. A ⟨character⟩ must be followed by a ⟨delimiter⟩ or by the end of the input. This rule resolves various ambiguous cases involving named characters, requiring, for example, the sequence of characters "`#\space`" to be interpreted as the space character rather than as the character "`#\s`" followed by the identifier "`pace`".

Note: The `#\newline` notation is retained for backward compatibility. Its use is deprecated; `#\linefeed` should be used instead.

4.2.7. Strings

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

- `\a` : alarm, U+0007
- `\b` : backspace, U+0008
- `\t` : character tabulation, U+0009
- `\n` : linefeed, U+000A
- `\v` : line tabulation, U+000B
- `\f` : formfeed, U+000C
- `\r` : return, U+000D
- `\"` : doublequote, U+0022
- `\\` : backslash, U+005C
- `\`⟨intraline whitespace⟩⟨line ending⟩
  ⟨intraline whitespace⟩ : nothing
- `\x`⟨hex scalar value⟩`;` : specified character (note the terminating semi-colon).

These escape sequences are case-sensitive, except that the alphabetic digits of a ⟨hex scalar value⟩ can be uppercase or lowercase.

Any other character in a string after a backslash is a syntax violation. Except for a line ending, any character outside of an escape sequence and not a doublequote stands for itself in the string literal. For example the single-character string literal `"λ"` (doublequote, a lower case lambda, doublequote) represents the same string as `"\x03bb;"`. A line ending that does not follow a backslash stands for a linefeed character.

Examples:

`"abc"`	U+0061, U+0062, U+0063
`"\x41;bc"`	`"Abc"` ; U+0041, U+0062, U+0063
`"\x41; bc"`	`"A bc"` U+0041, U+0020, U+0062, U+0063
`"\x41bc;"`	U+41BC
`"\x41"`	`&lexical` *exception*
`"\x;"`	`&lexical` *exception*
`"\x41bx;"`	`&lexical` *exception*
`"\x00000041;"`	`"A"` ; U+0041
`"\x0010FFFF;"`	U+10FFFF
`"\x00110000;"`	`&lexical` *exception* out of range
`"\x000000001;"`	U+0001
`"\xD800;"`	`&lexical` *exception* in excluded range
`"A` `bc"`	U+0041, U+000A, U+0062, U+0063 if no space occurs after the `A`

4.2.8. Numbers

The syntax of external representations for number objects is described formally by the ⟨number⟩ rule in the formal grammar. Case is not significant in external representations of number objects.

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

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

In systems with inexact number objects of varying precisions, it may be useful to specify the precision of a constant.

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

`3.1415926535898F0`
 Round to single, perhaps `3.141593`
`0.6L0`
 Extend to long, perhaps `.600000000000000`

A representation of a number object with nonempty mantissa width, $x|p$, represents the best binary floating-point approximation of x using a p-bit significand. For example, `1.1|53` is a representation of the best approximation of 1.1 in IEEE double precision. If x is an external representation of an inexact real number object that contains no vertical bar, then its numerical value should be computed as though it had a mantissa width of 53 or more.

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

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

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

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

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

If x is an external representation of an inexact real number object and contains no vertical bar and no exponent marker other than `e`, the inexact real number object it represents is a flonum (see library section 11.3). Some or all of the other external representations of inexact real number objects may also represent flonums, but that is not required by this report.

4.3. Datum syntax

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

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

- pairs and lists, enclosed by `( )` or `[ ]` (see section 4.3.2)
- vectors (see section 4.3.3)
- bytevectors (see section 4.3.4)

4.3.1. Formal account

The following grammar describes the syntax of syntactic data in terms of various kinds of lexemes defined in the grammar in section 4.2:

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

4.3.2. Pairs and lists

List and pair data, representing pairs and lists of values (see section 11.9) are represented using parentheses or brackets. Matching pairs of brackets that occur in the rules of ⟨list⟩ are equivalent to matching pairs of parentheses.

The most general notation for Scheme pairs as syntactic data is the "dotted" notation (⟨datum$_1$⟩ `.` ⟨datum$_2$⟩) where ⟨datum$_1$⟩ is the representation of the value of the car field and ⟨datum$_2$⟩ is the representation of the value of the cdr field. For example `(4 . 5)` is a pair whose car is 4 and whose cdr is 5.

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

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

and

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

are equivalent notations for a list of symbols.

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

The sequence of characters "`(4 . 5)`" is the external representation of a pair, not an expression that evaluates to a pair. Similarly, the sequence of characters "`(+ 2 6)`" is *not* an external representation of the integer 8, even though it *is* an expression (in the language of the `(rnrs base (6))` library) evaluating to the integer 8; rather, it is a syntactic datum representing a three-element list, the elements of which are the symbol `+` and the integers 2 and 6.

4.3.3. Vectors

Vector data, representing vectors of objects (see section 11.13), are represented using the notation `#(`⟨datum⟩ `...)`. For example, a vector of length 3 containing the number object for zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be represented as follows:

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

This is the external representation of a vector, not an expression that evaluates to a vector.

4.3.4. Bytevectors

Bytevector data, representing bytevectors (see library chapter 2), are represented using the notation `#vu8(`⟨u8⟩ `...)`, where the ⟨u8⟩s represent the octets of the bytevector. For example, a bytevector of length 3 containing the octets 2, 24, and 123 can be represented as follows:

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

This is the external representation of a bytevector, and also an expression that evaluates to a bytevector.

4.3.5. Abbreviations

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

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

5. Semantic concepts

5.1. Programs and libraries

A Scheme program consists of a *top-level program* together with a set of *libraries*, each of which defines a part of the program connected to the others through explicitly specified exports and imports. A library consists of a set of export and import specifications and a body, which consists of definitions, and expressions. A top-level program is similar to a library, but has no export specifications. Chapters 7 and 8 describe the syntax and semantics of libraries and top-level programs, respectively. Chapter 11 describes a base library that defines many of the constructs traditionally associated with Scheme. A separate report [24] describes the various *standard libraries* provided by a Scheme system.

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

5.2. Variables, keywords, and regions

Within the body of a library or top-level program, an identifier may name a kind of syntax, or it may name a location where a value can be stored. An identifier that names a kind of syntax is called a *keyword*, or *syntactic keyword*, and is said to be *bound* to that kind of syntax (or, in the case of a syntactic abstraction, a *transformer* that translates the syntax into more primitive forms; see section 9.2). An identifier that names a location is called a *variable* and is said to be *bound* to that location. At each point within a top-level program or a library, a specific, fixed set of identifiers is bound. The set of these identifiers, the set of *visible bindings*, is known as the *environment* in effect at that point.

Certain forms are used to create syntactic abstractions and to bind keywords to transformers for those new syntactic abstractions, while other forms create new locations and bind variables to those locations. Collectively, these forms are called *binding constructs*. Some binding constructs take the form of *definitions*, while others are expressions. With the exception of exported library bindings, a binding created by a definition is visible only within the body in which the definition appears, e.g., the body of a library, top-level program, or `lambda` expression. Exported library bindings are also visible within the bodies of the libraries and top-level programs that import them (see chapter 7).

Expressions that bind variables include the `lambda`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, and `let*-values` forms from the base library (see sections 11.4.2, 11.4.6). Of these, `lambda` is the most fundamental. Variable definitions appearing within the body of such an expression, or within the bodies of a library or top-level program, are treated as a set of `letrec*` bindings. In addition, for library bodies, the variables exported from the library can be referenced by importing libraries and top-level programs.

Expressions that bind keywords include the `let-syntax` and `letrec-syntax` forms (see section 11.18). A `define` form (see section 11.2.1) is a definition that creates a variable binding (see section 11.2), and a `define-syntax` form is a definition that creates a keyword binding (see section 11.2.2).

Scheme is a statically scoped language with block structure. To each place in a top-level program or library body where an identifier is bound there corresponds a *region* of code within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a `lambda` expression, for example, then its region is the entire `lambda` expression. Every mention of an identifier refers to the binding of the identifier that establishes the innermost of the regions containing the use. If a use of an identifier appears in a place where none of the surrounding expressions contains a binding for the identifier, the use may refer to a binding established by a definition or import at the top of the enclosing library or top-level program (see chapter 7). If there is no binding for the identifier, it is said to be *unbound*.

5.3. Exceptional situations

A variety of exceptional situations are distinguished in this report, among them violations of syntax, violations of a procedure's specification, violations of implementation restrictions, and exceptional situations in the environment. When an exceptional situation is detected by the implementation, an *exception is raised*, which means that a special procedure called the *current exception handler* is called. A program can also raise an exception, and override the current exception handler; see library section 7.1.

When an exception is raised, an object is provided that describes the nature of the exceptional situation. The report uses the condition system described in library section 7.2 to describe exceptional situations, classifying them by condition types.

Some exceptional situations allow continuing the program if the exception handler takes appropriate action. The corresponding exceptions are called *continuable*. For most of the exceptional situations described in this report, portable programs cannot rely upon the exception being continuable at the place where the situation was detected. For those exceptions, the exception handler that is invoked by the exception should not return. In some cases, however, continuing is permissible, and the handler may return. See library section 7.1.

Implementations must raise an exception when they are unable to continue correct execution of a correct program due to some *implementation restriction*. For example, an implementation that does not support infinities must raise an exception with condition type `&implementation-restriction` when it evaluates an expression whose result would be an infinity.

Some possible implementation restrictions such as the lack of representations for NaNs and infinities (see section 11.7.2) are anticipated by this report, and implementations typically must raise an exception of the appropriate condition type if they encounter such a situation.

This report uses the phrase "an exception is raised" synonymously with "an exception must be raised". This report uses the phrase "an exception with condition type *t*" to indicate that the object provided with the exception is a condition object of the specified type. The phrase "a continuable exception is raised" indicates an exceptional situation that permits the exception handler to return.

5.4. Argument checking

Many procedures specified in this report or as part of a standard library restrict the arguments they accept. Typically, a procedure accepts only specific numbers and types of arguments. Many syntactic forms similarly restrict the values to which one or more of their subforms can evaluate. These restrictions imply responsibilities for both the programmer and the implementation. Specifically, the programmer is responsible for ensuring that the values indeed adhere to the restrictions described in the specification. The implementation must check that the restrictions in the specification are indeed met, to the extent that it is reasonable, possible, and necessary to allow the specified operation to complete successfully. The implementation's responsibilities are specified in more detail in chapter 6 and throughout the report.

Note that it is not always possible for an implementation to completely check the restrictions set forth in a specification. For example, if an operation is specified to accept a procedure with specific properties, checking of these properties is undecidable in general. Similarly, some operations accept both lists and procedures that are called by these operations. Since lists can be mutated by the procedures through the `(rnrs mutable-pairs (6))` library (see library chapter 17), an argument that is a list when the operation starts may become a non-list during the execution of the operation. Also, the procedure might escape to a different continuation, preventing the operation from performing more checks. Requiring the operation to check that the argument is a list after each call to such a procedure would be impractical. Furthermore, some operations that accept lists only need to traverse these lists partially to perform their function; requiring the implementation to traverse the remainder of the list to verify that all specified restrictions have been met might violate reasonable performance assumptions. For these reasons, the programmer's obligations may exceed the checking obligations of the implementation.

When an implementation detects a violation of a restriction for an argument, it must raise an exception with condition type `&assertion` in a way consistent with the safety of execution as described in section 5.6.

5.5. Syntax violations

The subforms of a special form usually need to obey certain syntactic restrictions. As forms may be subject to macro expansion, which may not terminate, the question of whether they obey the specified restrictions is undecidable in general.

When macro expansion terminates, however, implementations must detect violations of the syntax. A *syntax violation* is an error with respect to the syntax of library bodies, top-level bodies, or the "syntax" entries in the specification of the base library or the standard libraries. Moreover, attempting to assign to an immutable variable (i.e., the variables exported by a library; see section 7.1) is also considered a syntax violation.

If a syntax violation occurs, the implementation must raise an exception with condition type `&syntax`, and execution of that top-level program or library must not be allowed to begin.

5.6. Safety

The standard libraries whose exports are described by this document are said to be *safe libraries*. Libraries and top-level programs that import only from safe libraries are also said to be safe.

As defined by this document, the Scheme programming language is safe in the following sense: The execution of a safe top-level program cannot go so badly wrong as to crash or to continue to execute while behaving in ways that are inconsistent with the semantics described in this document, unless an exception is raised.

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

The above safety properties are guaranteed only for top-level programs and libraries that are said to be safe. In particular, implementations may provide access to unsafe libraries in ways that cannot guarantee safety.

5.7. Boolean values

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

5.8. Multiple return values

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

Not all continuations accept any number of values. For example, a continuation that accepts the argument to a procedure call is guaranteed to accept exactly one value. The effect of passing some other number of values to such a continuation is unspecified. The `call-with-values` procedure described in section 11.15 makes it possible to create continuations that accept specified numbers of return values. If the number of return values passed to a continuation created by a call to `call-with-values` is not accepted by its consumer that was passed in that call, then an exception is raised. A more complete description of the number of values accepted by different continuations and the consequences of passing an unexpected number of values is given in the description of the `values` procedure in section 11.15.

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

5.9. Unspecified behavior

If an expression is said to "return unspecified values", then the expression must evaluate without raising an exception, but the values returned depend on the implementation; this report explicitly does not say how many or what values should be returned. Programmers should not rely on a specific number of return values or the specific values themselves.

5.10. Storage model

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

An object fetched from a location, by a variable reference or by a procedure such as `car`, `vector-ref`, or `string-ref`, is equivalent in the sense of `eqv?` (section 11.5) to the object last stored in the location before the fetch, except when that object is a procedure. When the object is a procedure, the object fetched from the location will also be a procedure that behaves identically to the procedure last stored in the location, but it is possible that it is not the same object.

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

It is desirable for constants (i.e. the values of literal expressions) to reside in read-only memory. To express this, it is convenient to imagine that every object that refers to locations is associated with a flag telling whether that object is mutable or immutable. Literal constants, the strings returned by `symbol->string`, records with no mutable fields, and other values explicitly designated as immutable are immutable objects, while all objects created by the other procedures listed in this report are mutable. An attempt to store a new value into a location referred to by an immutable object should raise an exception with condition type `&assertion`.

5.11. Proper tail recursion

Implementations of Scheme must be *properly tail-recursive*. Procedure calls that occur in certain syntactic contexts called *tail contexts* are *tail calls*. A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls. A call is *active* if the called procedure may still return. Note that this includes regular returns as well as returns through continuations captured earlier by `call-with-current-continuation` that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned. A formal definition of proper tail recursion can be found in Clinger's paper [5]. The rules for identifying tail calls in constructs from the `(rnrs base (6))` library are described in section 11.20.

5.12. Dynamic extent and the dynamic environment

For a procedure call, the time between when it is initiated and when it returns is called its *dynamic extent*. In Scheme, `call-with-current-continuation` (section 11.15) allows reentering a dynamic extent after its procedure call has returned. Thus, the dynamic extent of a call may not be a single, connected time period.

Some operations described in the report acquire information in addition to their explicit arguments from the *dynamic environment*. For example, `call-with-current-continuation` accesses an implicit context established by `dynamic-wind` (section 11.15), and the `raise` procedure (library section 7.1) accesses the current exception handler. The operations that modify the dynamic environment do so dynamically, for the dynamic extent of a call to a procedure like `dynamic-wind` or `with-exception-handler`. When such a call returns, the previous dynamic environment is restored. The dynamic environment can be thought of as part of the dynamic extent of a call. Consequently, it is captured by `call-with-current-continuation`, and restored by invoking the escape procedure it creates.

6. Entry format

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

template *category*

The *category* defines the kind of binding described by the entry, typically either “syntax” or “procedure”. An entry may specify various restrictions on subforms or arguments. For background on this, see section 5.4.

6.1. Syntax entries

If *category* is “syntax”, the entry describes a special syntactic construct, and the template gives the syntax of the forms of the construct. The template is written in a notation similar to a right-hand side of the BNF rules in chapter 4, and describes the set of forms equivalent to the forms matching the template as syntactic data. Some “syntax” entries carry a suffix (`expand`), specifying that the syntactic keyword of the construct is exported with level 1. Otherwise, the syntactic keyword is exported with level 0; see section 7.2.

Components of the form described by a template are designated by syntactic variables, which are written using angle brackets, for example, ⟨expression⟩, ⟨variable⟩. Case is insignificant in syntactic variables. Syntactic variables stand for other forms, or sequences of them. A syntactic variable may refer to a non-terminal in the grammar for syntactic data (see section 4.3.1), in which case only forms matching that non-terminal are permissible in that position. For example, ⟨identifier⟩ stands for a form which must be an identifier. Also, ⟨expression⟩ stands for any form which is a syntactically valid expression. Other non-terminals that are used in templates are defined as part of the specification.

The notation

⟨thing_1⟩ ...

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

⟨thing_1⟩ ⟨thing_2⟩ ...

indicates one or more occurrences of a ⟨thing⟩.

It is the programmer’s responsibility to ensure that each component of a form has the shape specified by a template. Descriptions of syntax may express other restrictions on the components of a form. Typically, such a restriction is formulated as a phrase of the form “⟨x⟩ must be a ...”. Again, these specify the programmer’s responsibility. It is the implementation’s responsibility to check that these restrictions are satisfied, as long as the macro transformers involved in expanding the form terminate. If the implementation detects that a component does not meet the restriction, an exception with condition type `&syntax` is raised.

6.2. Procedure entries

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

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

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

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

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

As with syntax templates, an ellipsis ... at the end of a header line, as in

(`=` z_1 z_2 z_3 ...) procedure

indicates that the procedure takes arbitrarily many arguments of the same type as specified for the last parameter name. In this case, `=` accepts two or more arguments that must all be complex number objects.

A procedure that detects an argument that it is not specified to handle must raise an exception with condition type `&assertion`. Also, the argument specifications are exhaustive: if the number of arguments provided in a procedure call does not match any number of arguments accepted by the procedure, an exception with condition type `&assertion` must be raised.

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

obj	any object
z	complex number object
x	real number object
y	real number object
q	rational number object
n	integer object
k	exact non-negative integer object
bool	boolean (`#f` or `#t`)
octet	exact integer object in $\{0, \ldots, 255\}$
byte	exact integer object in $\{-128, \ldots, 127\}$
char	character (see section 11.11)
pair	pair (see section 11.9)
vector	vector (see section 11.13)
string	string (see section 11.12)
condition	condition (see library section 7.2)
bytevector	bytevector (see library chapter 2)
proc	procedure (see section 1.6)

Other type restrictions are expressed through parameter-naming conventions that are described in specific chapters. For example, library chapter 11 uses a number of special parameter variables for the various subsets of the numbers.

With the listed type restrictions, it is the programmer's responsibility to ensure that the corresponding argument is of the specified type. It is the implementation's responsibility to check for that type.

A parameter called *list* means that it is the programmer's responsibility to pass an argument that is a list (see section 11.9). It is the implementation's responsibility to check that the argument is appropriately structured for the operation to perform its function, to the extent that this is possible and reasonable. The implementation must at least check that the argument is either an empty list or a pair.

Descriptions of procedures may express other restrictions on the arguments of a procedure. Typically, such a restriction is formulated as a phrase of the form "*x* must be a ..." (or otherwise using the word "must").

6.3. Implementation responsibilities

In addition to the restrictions implied by naming conventions, an entry may list additional explicit restrictions. These explicit restrictions usually describe both the programmer's responsibilities, who must ensure that the subforms of a form are appropriate, or that an appropriate argument is passed, and the implementation's responsibilities, which must check that subform adheres to the specified restrictions (if macro expansion terminates), or if the argument is appropriate. A description may explicitly list the implementation's responsibilities for some arguments or subforms in a paragraph labeled "*Implementation responsibilities*". In this case, the responsibilities specified for these subforms or arguments in the rest of the description are only for the programmer. A paragraph describing implementation responsibility does not affect the implementation's responsibilities for checking subforms or arguments not mentioned in the paragraph.

6.4. Other kinds of entries

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

`&who` condition type

indicates that `&who` is a condition type. The header line

`unquote` auxiliary syntax

indicates that `unquote` is a syntax binding that may occur only as part of specific surrounding expressions. Any use as an independent syntactic construct or identifier is a syntax violation. As with "syntax" entries, some "auxiliary syntax" entries carry a suffix (`expand`), specifying that the syntactic keyword of the construct is exported with level 1.

6.5. Equivalent entries

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

6.6. Evaluation examples

The symbol "⟹" used in program examples can be read "evaluates to". For example,

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

means that the expression `(* 5 8)` evaluates to the object `40`. Or, more precisely: the expression given by the sequence of characters "`(* 5 8)`" evaluates, in an environment that imports the relevant library, to an object that may be represented externally by the sequence of characters "`40`". See section 4.3 for a discussion of external representations of objects.

The "⟹" symbol is also used when the evaluation of an expression causes a violation. For example,

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

means that the evaluation of the expression `(integer->char #xD800)` must raise an exception with condition type `&assertion`.

Moreover, the "⟹" symbol is also used to explicitly say that the value of an expression in unspecified. For example:

```
(eqv? "" "")               ⟹  unspecified
```

Mostly, examples merely illustrate the behavior specified in the entry. In some cases, however, they disambiguate otherwise ambiguous specifications and are thus normative. Note that, in some cases, specifically in the case of inexact number objects, the return value is only specified conditionally or approximately. For example:

```
(atan -inf.0)
         ⟹ -1.5707963267948965 ; approximately
```

6.7. Naming conventions

By convention, the names of procedures that store values into previously allocated locations (see section 5.10) usually end in "`!`".

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

By convention, the names of predicates—procedures that always return a boolean value—end in "`?`" when the name contains any letters; otherwise, the predicate's name does not end with a question mark.

By convention, the components of compound names are separated by "`-`" In particular, prefixes that are actual words or can be pronounced as though they were actual words are followed by a hyphen, except when the first character following the hyphen would be something other than a letter, in which case the hyphen is omitted. Short, unpronounceable prefixes ("`fx`" and "`fl`") are not followed by a hyphen.

By convention, the names of condition types start with "`&`".

7. Libraries

Libraries are parts of a program that can be distributed independently. The library system supports macro definitions within libraries, macro exports, and distinguishes the phases in which definitions and imports are needed. This chapter defines the notation for libraries and a semantics for library expansion and execution.

7.1. Library form

A library definition must have the following form:

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

A library declaration contains the following elements:

- The ⟨library name⟩ specifies the name of the library (possibly with version).
- The `export` subform specifies a list of exports, which name a subset of the bindings defined within or imported into the library.
- The `import` subform specifies the imported bindings as a list of import dependencies, where each dependency specifies:
  - the imported library's name, and, optionally, constraints on its version,
  - the relevant levels, e.g., expand or run time (see section 7.2, and
  - the subset of the library's exports to make available within the importing library, and the local names to use within the importing library for each of the library's exports.
- The ⟨library body⟩ is the library body, consisting of a sequence of definitions followed by a sequence of expressions. The definitions may be both for local (unexported) and exported bindings, and the expressions are initialization expressions to be evaluated for their effects.

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

A ⟨library name⟩ uniquely identifies a library within an implementation, and is globally visible in the `import` clauses (see below) of all other libraries within an implementation. A ⟨library name⟩ has the following form:

(⟨identifier$_1$⟩ ⟨identifier$_2$⟩ ... ⟨version⟩)

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

(⟨sub-version⟩ ...)

Each ⟨sub-version⟩ must represent an exact nonnegative integer object. An empty ⟨version⟩ is equivalent to `()`.

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

⟨identifier⟩
`(rename` (⟨identifier$_1$⟩ ⟨identifier$_2$⟩) ...)

In an ⟨export spec⟩, an ⟨identifier⟩ names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A `rename` spec exports the binding named by ⟨identifier$_1$⟩ in each (⟨identifier$_1$⟩ ⟨identifier$_2$⟩) pairing, using ⟨identifier$_2$⟩ as the external name.

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

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

An ⟨import level⟩ is one of the following:

`run`
`expand`
`(meta` ⟨level⟩`)`

where ⟨level⟩ represents an exact integer object.

As an ⟨import level⟩, `run` is an abbreviation for `(meta 0)`, and `expand` is an abbreviation for `(meta 1)`. Levels and phases are discussed in section 7.2.

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

⟨library reference⟩
`(library` ⟨library reference⟩`)`
`(only` ⟨import set⟩ ⟨identifier⟩ `...)`
`(except` ⟨import set⟩ ⟨identifier⟩ `...)`
`(prefix` ⟨import set⟩ ⟨identifier⟩`)`
`(rename` ⟨import set⟩ `(`⟨identifier$_1$⟩ ⟨identifier$_2$⟩`) ...)`

A ⟨library reference⟩ identifies a library by its name and optionally by its version. It has one of the following forms:

`(`⟨identifier$_1$⟩ ⟨identifier$_2$⟩ `...)`
`(`⟨identifier$_1$⟩ ⟨identifier$_2$⟩ `...` ⟨version reference⟩`)`

A ⟨library reference⟩ whose first ⟨identifier⟩ is `for`, `library`, `only`, `except`, `prefix`, or `rename` is permitted only within a `library` ⟨import set⟩. The ⟨import set⟩ `(library` ⟨library reference⟩`)` is otherwise equivalent to ⟨library reference⟩.

A ⟨library reference⟩ with no ⟨version reference⟩ (first form above) is equivalent to a ⟨library reference⟩ with a ⟨version reference⟩ of `()`.

A ⟨version reference⟩ specifies a set of ⟨version⟩s that it matches. The ⟨library reference⟩ identifies all libraries of the same name and whose version is matched by the ⟨version reference⟩. A ⟨version reference⟩ has the following form:

`(`⟨sub-version reference$_1$⟩ `...` ⟨sub-version reference$_n$⟩`)`
`(and` ⟨version reference⟩ `...)`
`(or` ⟨version reference⟩ `...)`
`(not` ⟨version reference⟩`)`

A ⟨version reference⟩ of the first form matches a ⟨version⟩ with at least n elements, whose ⟨sub-version reference⟩s match the corresponding ⟨sub-version⟩s. An `and` ⟨version reference⟩ matches a version if all ⟨version references⟩ following the `and` match it. Correspondingly, an `or` ⟨version reference⟩ matches a version if one of ⟨version references⟩ following the `or` matches it, and a `not` ⟨version reference⟩ matches a version if the ⟨version reference⟩ following it does not match it.

A ⟨sub-version reference⟩ has one of the following forms:

⟨sub-version⟩
`(>=` ⟨sub-version⟩`)`
`(<=` ⟨sub-version⟩`)`
`(and` ⟨sub-version reference⟩ `...)`
`(or` ⟨sub-version reference⟩ `...)`
`(not` ⟨sub-version reference⟩`)`

A ⟨sub-version reference⟩ of the first form matches a ⟨sub-version⟩ if it is equal to it. A `>=` ⟨sub-version reference⟩ form matches a sub-version if it is greater or equal to the ⟨sub-version⟩ following it; analogously for `<=`. An `and` ⟨sub-version reference⟩ matches a sub-version if all of the subsequent ⟨sub-version reference⟩s match it. Correspondingly, an `or` ⟨sub-version reference⟩ matches a sub-version if one of the subsequent ⟨sub-version reference⟩s matches it, and a `not` ⟨sub-version reference⟩ matches a sub-version if the subsequent ⟨sub-version reference⟩ does not match it.

Examples:

version reference	version	match?
`()`	`(1)`	yes
`(1)`	`(1)`	yes
`(1)`	`(2)`	no
`(2 3)`	`(2)`	no
`(2 3)`	`(2 3)`	yes
`(2 3)`	`(2 3 5)`	yes
`(or (1 (>= 1)) (2))`	`(2)`	yes
`(or (1 (>= 1)) (2))`	`(1 1)`	yes
`(or (1 (>= 1)) (2))`	`(1 0)`	no
`((or 1 2 3))`	`(1)`	yes
`((or 1 2 3))`	`(2)`	yes
`((or 1 2 3))`	`(3)`	yes
`((or 1 2 3))`	`(4)`	no

When more than one library is identified by a library reference, the choice of libraries is determined in some implementation-dependent manner.

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

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

- An `only` form produces a subset of the bindings from another ⟨import set⟩, including only the listed ⟨identifier⟩s. The included ⟨identifier⟩s must be in the original ⟨import set⟩.

- An `except` form produces a subset of the bindings from another ⟨import set⟩, including all but the listed ⟨identifier⟩s. All of the excluded ⟨identifier⟩s must be in the original ⟨import set⟩.
- A `prefix` form adds the ⟨identifier⟩ prefix to each name from another ⟨import set⟩.
- A `rename` form, `(rename (`⟨identifier$_1$⟩ ⟨identifier$_2$⟩`) ...)`, removes the bindings for ⟨identifier$_1$⟩ `...` to form an intermediate ⟨import set⟩, then adds the bindings back for the corresponding ⟨identifier$_2$⟩ `...` to form the final ⟨import set⟩. Each ⟨identifier$_1$⟩ must be in the original ⟨import set⟩, each ⟨identifier$_2$⟩ must not be in the intermediate ⟨import set⟩, and the ⟨identifier$_2$⟩s must be distinct.

It is a syntax violation if a constraint given above is not met.

The ⟨library body⟩ of a `library` form consists of forms that are classified as *definitions* or *expressions*. Which forms belong to which class depends on the imported libraries and the result of expansion—see chapter 10. Generally, forms that are not definitions (see section 11.2 for definitions available through the base library) are expressions.

A ⟨library body⟩ is like a ⟨body⟩ (see section 11.3) except that a ⟨library body⟩s need not include any expressions. It must have the following form:

⟨definition⟩ ... ⟨expression⟩ ...

When `begin`, `let-syntax`, or `letrec-syntax` forms occur in a library body prior to the first expression, they are spliced into the body; see section 11.4.7. Some or all of the body, including portions wrapped in `begin`, `let-syntax`, or `letrec-syntax` forms, may be specified by a syntactic abstraction (see section 9.2).

The transformer expressions and bindings are evaluated and created from left to right, as described in chapter 10. The expressions of variable definitions are evaluated from left to right, as if in an implicit `letrec*`, and the body expressions are also evaluated from left to right after the expressions of the variable definitions. A fresh location is created for each exported variable and initialized to the value of its local counterpart. The effect of returning twice to the continuation of the last body expression is unspecified.

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

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

All explicitly exported variables are immutable in both the exporting and importing libraries. It is thus a syntax violation if an explicitly exported variable appears on the left-hand side of a `set!` expression, either in the exporting or importing libraries.

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

All other variables defined within a library are mutable.

7.2. Import and export levels

Expanding a library may require run-time information from another library. For example, if a macro transformer calls a procedure from library A, then the library A must be instantiated before expanding any use of the macro in library B. Library A may not be needed when library B is eventually run as part of a program, or it may be needed for run time of library B, too. The library mechanism distinguishes these times by phases, which are explained in this section.

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

A *phase* is a time at which the expressions within a library are evaluated. Within a library body, top-level expressions and the right-hand sides of `define` forms are evaluated at run time, i.e., phase 0, and the right-hand sides of `define-syntax` forms are evaluated at expand time, i.e., phase 1. When `define-syntax`, `let-syntax`, or `letrec-syntax` forms appear within code evaluated at phase n, the right-hand sides are evaluated at phase $n+1$.

These phases are relative to the phase in which the library itself is used. An *instance* of a library corresponds to an evaluation of its variable definitions and expressions in a particular phase relative to another library—a process called *instantiation*. For example, if a top-level expression in a library B refers to a variable export from another library A, then it refers to the export from an instance of A at phase 0 (relative to the phase of B). But if a phase 1 expression within B refers to the same binding from A, then it refers to the export from an instance of A at phase 1 (relative to the phase of B).

A *visit* of a library corresponds to the evaluation of its syntax definitions in a particular phase relative to another library—a process called *visiting*. For example, if a top-level expression in a library B refers to a macro export from another library A, then it refers to the export from a visit of A at phase 0 (relative to the phase of B), which corresponds to the evaluation of the macro's transformer expression at phase 1.

A *level* is a lexical property of an identifier that determines in which phases it can be referenced. The level for each identifier bound by a definition within a library is 0; that is, the identifier can be referenced only at phase 0 within the library. The level for each imported binding is determined by the enclosing `for` form of the `import` in the importing library, in addition to the levels of the identifier in the exporting library. Import and export levels are combined by pairwise addition of all level combinations. For example, references to an imported identifier exported for levels p_a and p_b and imported for levels q_a, q_b, and q_c are valid at levels p_a+q_a, p_a+q_b, p_a+q_c, p_b+q_a, p_b+q_b, and p_b+q_c. An ⟨import set⟩ without an enclosing `for` is equivalent to `(for` ⟨import set⟩ `run)`, which is the same as `(for` ⟨import set⟩ `(meta 0))`.

The export level of an exported binding is 0 for all bindings that are defined within the exporting library. The export levels of a reexported binding, i.e., an export imported from another library, are the same as the effective import levels of that binding within the reexporting library.

For the libraries defined in the library report, the export level is 0 for nearly all bindings. The exceptions are `syntax-rules`, `identifier-syntax`, `...`, and `_` from the `(rnrs base (6))` library, which are exported with level 1, `set!` from the `(rnrs base (6))` library, which is exported with levels 0 and 1, and all bindings from the composite `(rnrs (6))` library (see library chapter 15), which are exported with levels 0 and 1.

Macro expansion within a library can introduce a reference to an identifier that is not explicitly imported into the library. In that case, the phase of the reference must match the identifier's level as shifted by the difference between the phase of the source library (i.e., the library that supplied the identifier's lexical context) and the library that encloses the reference. For example, suppose that expanding a library invokes a macro transformer, and the evaluation of the macro transformer refers to an identifier that is exported from another library (so the phase-1 instance of the library is used); suppose further that the value of the binding is a syntax object representing an identifier with only a level-n binding; then, the identifier must be used only at phase $n+1$ in the library being expanded. This combination of levels and phases is why negative levels on identifiers can be useful, even though libraries exist only at non-negative phases.

If any of a library's definitions are referenced at phase 0 in the expanded form of a program, then an instance of the referenced library is created for phase 0 before the program's definitions and expressions are evaluated. This rule applies transitively: if the expanded form of one library references at phase 0 an identifier from another library, then before the referencing library is instantiated at phase n, the referenced library must be instantiated at phase n. When an identifier is referenced at any phase n greater than 0, in contrast, then the defining library is instantiated at phase n at some unspecified time before the reference is evaluated. Similarly, when a macro keyword is referenced at phase n during the expansion of a library, then the defining library is visited at phase n at some unspecified time before the reference is evaluated.

An implementation may distinguish instances/visits of a library for different phases or to use an instance/visit at any phase as an instance/visit at any other phase. An implementation may further expand each `library` form with distinct visits of libraries in any phase and/or instances of libraries in phases above 0. An implementation may create instances/visits of more libraries at more phases than required to satisfy references. When an identifier appears as an expression in a phase that is inconsistent with the identifier's level, then an implementation may raise an exception either at expand time or run time, or it may allow the reference. Thus, a library whose meaning depends on whether the instances of a library are distinguished or shared across phases or `library` expansions may be unportable.

7.3. Examples

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

```
(library (stack)
  (export make push! pop! empty!)
  (import (rnrs)
          (rnrs mutable-pairs))

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

```
  (define (empty! s) (set-car! s '())))

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

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

(library (party)
  ;; Total exports:
  ;; make, push, push!, make-party, pop!
  (export (rename (balloon:make make)
                  (balloon:push push))
          push!
          make-party
          (rename (party-pop! pop!)))
  (import (rnrs)
          (only (stack) make push! pop!) ; not empty!
          (prefix (balloons) balloon:))

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


(library (main)
  (export)
  (import (rnrs) (party))

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

Examples for macros and phases:

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

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

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

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

(library (let-div)
  (export let-div)
  (import (rnrs)
          (my-helpers values-stuff)
          (rnrs r5rs))

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

8. Top-level programs

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

8.1. Top-level program syntax

A top-level program is a delimited piece of text, typically a file, that has the following form:

$\langle$import form$\rangle$ $\langle$top-level body$\rangle$

An $\langle$import form$\rangle$ has the following form:

`(import` $\langle$import spec$\rangle$ `...)`

A $\langle$top-level body$\rangle$ has the following form:

$\langle$top-level body form$\rangle$ ...

A $\langle$top-level body form$\rangle$ is either a $\langle$definition$\rangle$ or an $\langle$expression$\rangle$.

The $\langle$import form$\rangle$ is identical to the import clause in libraries (see section 7.1), and specifies a set of libraries to import. A $\langle$top-level body$\rangle$ is like a $\langle$library body$\rangle$ (see section 7.1), except that definitions and expressions may occur in any order. Thus, the syntax specified by

⟨top-level body form⟩ refers to the result of macro expansion.

When uses of `begin`, `let-syntax`, or `letrec-syntax` from the `(rnrs base (6))` library occur in a top-level body prior to the first expression, they are spliced into the body; see section 11.4.7. Some or all of the body, including portions wrapped in `begin`, `let-syntax`, or `letrec-syntax` forms, may be specified by a syntactic abstraction (see section 9.2).

8.2. Top-level program semantics

A top-level program is executed by treating the program similarly to a library, and evaluating its definitions and expressions. The semantics of a top-level body may be roughly explained by a simple translation into a library body: Each ⟨expression⟩ that appears before a definition in the top-level body is converted into a dummy definition

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

where ⟨variable⟩ is a fresh identifier and ⟨unspecified⟩ is a side-effect-free expression returning an unspecified value. (It is generally impossible to determine which forms are definitions and expressions without concurrently expanding the body, so the actual translation is somewhat more complicated; see chapter 10.)

On platforms that support it, a top-level program may access its command line by calling the `command-line` procedure (see library section 10).

9. Primitive syntax

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

9.1. Primitive expression types

The entries in this section all describe expressions, which may occur in the place of ⟨expression⟩ syntactic variables. See also section 11.4.

Constant literals

⟨number⟩ syntax
⟨boolean⟩ syntax
⟨character⟩ syntax
⟨string⟩ syntax
⟨bytevector⟩ syntax

An expression consisting of a representation of a number object, a boolean, a character, a string, or a bytevector, evaluates "to itself".

```
145932                ⟹  145932
#t                    ⟹  #t
"abc"                 ⟹  "abc"
#vu8(2 24 123)        ⟹  #vu8(2 24 123)
```

As noted in section 5.10, the value of a literal expression is immutable.

Variable references

⟨variable⟩ syntax

An expression consisting of a variable (section 5.2) is a variable reference if it is not a macro use (see below). The value of the variable reference is the value stored in the location to which the variable is bound. It is a syntax violation to reference an unbound variable.

The following example examples assumes the base library has been imported:

```
(define x 28)
x                     ⟹  28
```

Procedure calls

(⟨operator⟩ ⟨operand$_1$⟩ ...) syntax

A procedure call consists of expressions for the procedure to be called and the arguments to be passed to it, with enclosing parentheses. A form in an expression context is a procedure call if ⟨operator⟩ is not an identifier bound as a syntactic keyword (see section 9.2 below).

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

The following examples assume the `(rnrs base (6))` library has been imported:

```
(+ 3 4)               ⟹  7
((if #f + *) 3 4)     ⟹  12
```

If the value of ⟨operator⟩ is not a procedure, an exception with condition type `&assertion` is raised. Also, if ⟨operator⟩ does not accept as many arguments as there are ⟨operand⟩s, an exception with condition type `&assertion` is raised.

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

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

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

9.2. Macros

Libraries and top-level programs can define and use new kinds of derived expressions and definitions called *syntactic abstractions* or *macros*. A syntactic abstraction is created by binding a keyword to a *macro transformer* or, simply, *transformer*. The transformer determines how a use of the macro (called a *macro use*) is transcribed into a more primitive form.

Most macro uses have the form:

(⟨keyword⟩ ⟨datum⟩ ...)

where ⟨keyword⟩ is an identifier that uniquely determines the kind of form. This identifier is called the *syntactic keyword*, or simply *keyword*, of the macro. The number of ⟨datum⟩s and the syntax of each depends on the syntactic abstraction.

Macro uses can also take the form of improper lists, singleton identifiers, or `set!` forms, where the second subform of the `set!` is the keyword (see section 11.19) library section 12.3):

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

The `define-syntax`, `let-syntax` and `letrec-syntax` forms, described in sections 11.2.2 and 11.18, create bindings for keywords, associate them with macro transformers, and control the scope within which they are visible.

The `syntax-rules` and `identifier-syntax` forms, described in section 11.19, create transformers via a pattern language. Moreover, the `syntax-case` form, described in library chapter 12, allows creating transformers via arbitrary Scheme code.

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

Macros defined using `syntax-rules` and `identifier-syntax` are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping [16, 15, 2, 6, 9]:

- If a macro transformer inserts a binding for an identifier (variable or keyword) not appearing in the macro use, the identifier is in effect renamed throughout its scope to avoid conflicts with other identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

Macros defined using the `syntax-case` facility are also hygienic unless `datum->syntax` (see library section 12.6) is used.

10. Expansion process

Macro uses (see section 9.2) are expanded into *core forms* at the start of evaluation (before compilation or interpretation) by a syntax *expander*. The set of core forms is implementation-dependent, as is the representation of these forms in the expander's output. If the expander encounters a syntactic abstraction, it invokes the associated transformer to expand the syntactic abstraction, then repeats the expansion process for the form returned by the transformer. If the expander encounters a core form, it recursively processes its subforms that are in expression or definition context, if any, and reconstructs the form from the expanded subforms. Information about identifier bindings is maintained during expansion to enforce lexical scoping for variables and keywords.

To handle definitions, the expander processes the initial forms in a ⟨body⟩ (see section 11.3) or ⟨library body⟩ (see section 7.1) from left to right. How the expander processes each form encountered depends upon the kind of form.

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

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

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

`begin` **form** The expander splices the subforms into the list of body forms it is processing. (See section 11.4.7.)

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

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

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

A definition in the sequence of forms must not define any identifier whose binding is used to determine the meaning of the undeferred portions of the definition or any definition that precedes it in the sequence of forms. For example, the bodies of the following expressions violate this restriction.

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

(let-syntax ([def0 (syntax-rules ()
                     [(_ x) (define x 0)])])
  (let ([z 3])
    (def0 z)
    (define def0 list)
    (list z)))

(let ()
  (define-syntax foo
    (lambda (e)
      (+ 1 2)))
  (define + 2)
  (foo))
```

The following do not violate the restriction.

```
(let ([x 5])
  (define lambda list)
  (lambda x x))              ⟹  (5 5)

(let-syntax ([def0 (syntax-rules ()
                     [(_ x) (define x 0)])])
  (let ([z 3])
    (define def0 list)
    (def0 z)
    (list z)))               ⟹  (3)

(let ()
  (define-syntax foo
    (lambda (e)
      (let ([+ -]) (+ 1 2))))
  (define + 2)
  (foo))                     ⟹  -1
```

The implementation should treat a violation of the restriction as a syntax violation.

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

Example:

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

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

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

although the structure of the output is implementation-dependent.

Because definitions and expressions can be interleaved in a ⟨top-level body⟩ (see chapter 8), the expander's processing of a ⟨top-level body⟩ is somewhat more complicated. It behaves as described above for a ⟨body⟩ or ⟨library body⟩ with the following exceptions: When the expander finds a nondefinition, it defers its expansion and continues scanning for definitions. Once it reaches the end of the set of forms, it processes the deferred right-hand-side and body expressions, then generates the equivalent of a `letrec*` form from the defined variables, expanded right-hand-side expressions, and expanded body expressions. For each body expression ⟨expression⟩ that appears before a variable definition in the body, a dummy binding is created at the corresponding place within the set of `letrec*` bindings, with a fresh temporary variable on the left-hand side and the equivalent of (begin ⟨expression⟩ ⟨unspecified⟩), where ⟨unspecified⟩ is a side-effect-free expression returning an unspecified value, on the right-hand side, so that left-to-right evaluation order is preserved. The `begin` wrapper allows ⟨expression⟩ to evaluate to an arbitrary number of values.

11. Base library

This chapter describes Scheme's `(rnrs base (6))` library, which exports many of the procedure and syntax bindings that are traditionally associated with Scheme.

Section 11.20 defines the rules that identify tail calls and tail contexts in constructs from the `(rnrs base (6))` library.

11.1. Base types

No object satisfies more than one of the following predicates:

```
boolean?        pair?
symbol?         number?
char?           string?
vector?         procedure?
null?
```

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

Note that, although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test; see section 5.7.

11.2. Definitions

Definitions may appear within a ⟨top-level body⟩ (section 8.1), at the top of a ⟨library body⟩ (section 7.1), or at the top of a ⟨body⟩ (section 11.3).

A ⟨definition⟩ may be a variable definition (section 11.2.1) or keyword definition (section 11.2.1). Macro uses that expand into definitions or groups of definitions (packaged in a `begin`, `let-syntax`, or `letrec-syntax` form; see section 11.4.7) may also appear wherever other definitions may appear.

11.2.1. Variable definitions

The `define` form described in this section is a ⟨definition⟩ used to create variable bindings and may appear anywhere other definitions may appear.

(define ⟨variable⟩ ⟨expression⟩) syntax
(define ⟨variable⟩) syntax
(define (⟨variable⟩ ⟨formals⟩) ⟨body⟩) syntax
(define (⟨variable⟩ . ⟨formal⟩) ⟨body⟩) syntax

The first from of `define` binds ⟨variable⟩ to a new location before assigning the value of ⟨expression⟩ to it.

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

The continuation of ⟨expression⟩ should not be invoked more than once.

Implementation responsibilities: Implementations should detect that the continuation of ⟨expression⟩ is invoked more than once. If the implementation detects this, it must raise an exception with condition type `&assertion`.

The second form of `define` is equivalent to

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

where ⟨unspecified⟩ is a side-effect-free expression returning an unspecified value.

In the third form of `define`, ⟨formals⟩ must be either a sequence of zero or more variables, or a sequence of one or more variables followed by a dot `.` and another variable (as in a lambda expression, see section 11.4.2). This form is equivalent to

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

In the fourth form of `define`, ⟨formal⟩ must be a single variable. This form is equivalent to

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

11.2.2. Syntax definitions

The `define-syntax` form described in this section is a ⟨definition⟩ used to create keyword bindings and may appear anywhere other definitions may appear.

(define-syntax ⟨keyword⟩ ⟨expression⟩) syntax

Binds ⟨keyword⟩ to the value of ⟨expression⟩, which must evaluate, at macro-expansion time, to a transformer. Macro transformers can be created using the `syntax-rules` and `identifier-syntax` forms described in section 11.19. See library section 12.3 for a more complete description of transformers.

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

Example:

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

Implementation responsibilities: The implementation should detect if the value of ⟨expression⟩ cannot possibly be a transformer.

An implication of the left-to-right processing order (section 10) is that one definition can affect whether a subsequent form is also a definition.

Example:

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

The behavior is unaffected by any binding for `bind-to-zero` that might appear outside of the `let` expression.

11.3. Bodies

The ⟨body⟩ of a `lambda`, `let`, `let*`, `let-values`, `let*-values`, `letrec`, or `letrec*` expression, or that of a definition with a body consists of zero or more definitions followed by one or more expressions.

⟨definition⟩ ... ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...

Each identifier defined by a definition is local to the ⟨body⟩. That is, the identifier is bound, and the region of the binding is the entire ⟨body⟩ (see section 5.2).

Example:

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

When `begin`, `let-syntax`, or `letrec-syntax` forms occur in a body prior to the first expression, they are spliced into the body; see section 11.4.7. Some or all of the body, including portions wrapped in `begin`, `let-syntax`, or `letrec-syntax` forms, may be specified by a macro use (see section 9.2).

An expanded ⟨body⟩ (see chapter 10) containing variable definitions can always be converted into an equivalent `letrec*` expression. For example, the `let` expression in the above example is equivalent to

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

11.4. Expressions

The entries in this section describe the expressions of the `(rnrs base (6))` library, which may occur in the position of the ⟨expression⟩ syntactic variable in addition to the primitive expression types as described in section 9.1.

11.4.1. Quotation

`(quote` ⟨datum⟩`)` syntax

Syntax: ⟨Datum⟩ should be a syntactic datum.

Semantics: `(quote` ⟨datum⟩`)` evaluates to the datum value represented by ⟨datum⟩ (see section 4.3). This notation is used to include constants.

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

As noted in section 4.3.5, `(quote` ⟨datum⟩`)` may be abbreviated as `'`⟨datum⟩:

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

As noted in section 5.10, constants are immutable.

Note: Different constants that are the value of a `quote` expression may share the same locations.

11.4.2. Procedures

`(lambda` ⟨formals⟩ ⟨body⟩`)` syntax

Syntax: ⟨Formals⟩ must be a formal parameter list as described below, and ⟨body⟩ must be as described in section 11.3.

Semantics: A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression is evaluated is remembered as part of the procedure. When the procedure is later called with some arguments, the environment in which the `lambda` expression was evaluated is extended by binding the variables in the parameter list to fresh locations, and the resulting argument values are stored in those locations. Then, the expressions in the body of the `lambda` expression (which may contain definitions and thus represent a `letrec*` form, see section 11.3) are evaluated sequentially in the extended environment. The results of the last expression in the body are returned as the results of the procedure call.

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

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

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

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

⟨Formals⟩ must have one of the following forms:

- `(`⟨variable$_1$⟩ `...`): The procedure takes a fixed number of arguments; when the procedure is called, the arguments are stored in the bindings of the corresponding variables.
- ⟨variable⟩: The procedure takes any number of arguments; when the procedure is called, the sequence of arguments is converted into a newly allocated list, and the list is stored in the binding of the ⟨variable⟩.
- `(`⟨variable$_1$⟩ `...` ⟨variable$_n$⟩ `.` ⟨variable$_{n+1}$⟩`)`: If a period `.` precedes the last variable, then the procedure takes n or more arguments, where n is the number of parameters before the period (there must be at least one). The value stored in the binding of the last variable is a newly allocated list of the arguments left over after all the other arguments have been matched up against the other parameters.

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

Any ⟨variable⟩ must not appear more than once in ⟨formals⟩.

11.4.3. Conditionals

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

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

Semantics: An `if` expression is evaluated as follows: first, ⟨test⟩ is evaluated. If it yields a true value (see section 5.7), then ⟨consequent⟩ is evaluated and its values are returned. Otherwise ⟨alternate⟩ is evaluated and its values are returned. If ⟨test⟩ yields `#f` and no ⟨alternate⟩ is specified, then the result of the expression is unspecified.

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

The ⟨consequent⟩ and ⟨alternate⟩ expressions are in tail context if the `if` expression itself is; see section 11.20.

11.4.4. Assignments

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

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

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

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

Note: The identifier `set!` is exported with level 1 as well. See section 11.19.

11.4.5. Derived conditionals

`(cond` ⟨cond clause$_1$⟩ ⟨cond clause$_2$⟩ `...)` syntax
`=>` auxiliary syntax
`else` auxiliary syntax

Syntax: Each ⟨cond clause⟩ must be of the form

`(`⟨test⟩ ⟨expression$_1$⟩ `...)`

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

`(`⟨test⟩ `=>` ⟨expression⟩`)`

The last ⟨cond clause⟩ may be an "`else` clause", which has the form (else ⟨expression$_1$⟩ ⟨expression$_2$⟩ . . .).

Semantics: A `cond` expression is evaluated by evaluating the ⟨test⟩ expressions of successive ⟨cond clause⟩s in order until one of them evaluates to a true value (see section 5.7). When a ⟨test⟩ evaluates to a true value, then the remaining ⟨expression⟩s in its ⟨cond clause⟩ are evaluated in order, and the results of the last ⟨expression⟩ in the ⟨cond clause⟩ are returned as the results of the entire `cond` expression. If the selected ⟨cond clause⟩ contains only the ⟨test⟩ and no ⟨expression⟩s, then the value of the ⟨test⟩ is returned as the result. If the selected ⟨cond clause⟩ uses the `=>` alternate form, then the ⟨expression⟩ is evaluated. Its value must be a procedure. This procedure should accept one argument; it is called on the value of the ⟨test⟩ and the values returned by this procedure are returned by the `cond` expression. If all ⟨test⟩s evaluate to `#f`, and there is no `else` clause, then the conditional expression returns unspecified values; if there is an `else` clause, then its ⟨expression⟩s are evaluated, and the values of the last one are returned.

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

For a ⟨cond clause⟩ of one of the following forms

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

the last ⟨expression⟩ is in tail context if the `cond` form itself is. For a ⟨cond clause⟩ of the form

(⟨test⟩ => ⟨expression⟩)

the (implied) call to the procedure that results from the evaluation of ⟨expression⟩ is in a tail context if the `cond` form itself is. See section 11.20.

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

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

Syntax: ⟨Key⟩ must be an expression. Each ⟨case clause⟩ must have one of the following forms:

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

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

Semantics: A `case` expression is evaluated as follows. ⟨Key⟩ is evaluated and its result is compared using `eqv?` (see section 11.5) against the data represented by the ⟨datum⟩s of each ⟨case clause⟩ in turn, proceeding in order from left to right through the set of clauses. If the result of evaluating ⟨key⟩ is equivalent to a datum of a ⟨case clause⟩, the corresponding ⟨expression⟩s are evaluated from left to right and the results of the last expression in the ⟨case clause⟩ are returned as the results of the `case` expression. Otherwise, the comparison process continues. If the result of evaluating ⟨key⟩ is different from every datum in each set, then if there is an `else` clause its expressions are evaluated and the results of the last are the results of the `case` expression; otherwise the `case` expression returns unspecified values.

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

The last ⟨expression⟩ of a ⟨case clause⟩ is in tail context if the `case` expression itself is; see section 11.20.

(and ⟨test$_1$⟩ . . .) syntax

Syntax: The ⟨test⟩s must be expressions.

Semantics: If there are no ⟨test⟩s, `#t` is returned. Otherwise, the ⟨test⟩ expressions are evaluated from left to right until a ⟨test⟩ returns `#f` or the last ⟨test⟩ is reached. In the former case, the `and` expression returns `#f` without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

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

The `and` keyword could be defined in terms of `if` using `syntax-rules` (see section 11.19) as follows:

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

The last ⟨test⟩ expression is in tail context if the `and` expression itself is; see section 11.20.

(or ⟨test$_1$⟩ . . .) syntax

Syntax: The ⟨test⟩s must be expressions.

Semantics: If there are no ⟨test⟩s, `#f` is returned. Otherwise, the ⟨test⟩ expressions are evaluated from left to right until a ⟨test⟩ returns a true value *val* (see section 5.7) or the last ⟨test⟩ is reached. In the former case, the `or` expression returns *val* without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

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

The `or` keyword could be defined in terms of `if` using `syntax-rules` (see section 11.19) as follows:

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

The last ⟨test⟩ expression is in tail context if the `or` expression itself is; see section 11.20.

11.4.6. Binding constructs

The binding constructs described in this section create local bindings for variables that are visible only in a delimited region. The syntax of the constructs `let`, `let*`, `letrec`, and `letrec*` is identical, but they differ in the regions (see section 5.2) they establish for their variable bindings and in the order in which the values for the bindings are computed. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially. In a `letrec` or `letrec*` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. In a `letrec` expression, the initial values are computed before being assigned to the variables; in a `letrec*`, the evaluations and assignments are performed sequentially.

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

Sample definitions of all the binding forms of this section in terms of simpler forms are in appendix B.

`(let` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 11.3. Any variable must not appear more than once in the ⟨variable⟩s.

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

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

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

See also named `let`, section 11.16.

`(let*` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 11.3.

Semantics: The `let*` form is similar to `let`, but the ⟨init⟩s are evaluated and bindings created sequentially from left to right, with the region of each binding including the bindings to its right as well as ⟨body⟩. Thus the second ⟨init⟩ is evaluated in an environment in which the first binding is visible and initialized, and so on.

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

Note: While the variables bound by a `let` expression must be distinct, the variables bound by a `let*` expression need not be distinct.

`(letrec` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 11.3. Any variable must not appear more than once in the ⟨variable⟩s.

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

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

It should be possible to evaluate each ⟨init⟩ without assigning or referring to the value of any ⟨variable⟩. In the most common uses of `letrec`, all the ⟨init⟩s are `lambda` expressions and the restriction is satisfied automatically. Another restriction is that the continuation of each ⟨init⟩ should not be invoked more than once.

Implementation responsibilities: Implementations must detect references to a ⟨variable⟩ during the evaluation of the ⟨init⟩ expressions (using one particular evaluation order and order of evaluating the ⟨init⟩ expressions). If an implementation detects such a violation of the restriction, it must raise an exception with condition type `&assertion`. Implementations may or may not detect that the continuation of each ⟨init⟩ is invoked more than once. However, if the implementation detects this, it must raise an exception with condition type `&assertion`.

(letrec* ⟨bindings⟩ ⟨body⟩) syntax

Syntax: ⟨Bindings⟩ must have the form

(($\langle\text{variable}_1\rangle$ $\langle\text{init}_1\rangle$) ...),

where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 11.3. Any variable must not appear more than once in the ⟨variable⟩s.

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

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

It must be possible to evaluate each ⟨init⟩ without assigning or referring to the value of the corresponding ⟨variable⟩ or the ⟨variable⟩ of any of the bindings that follow it in ⟨bindings⟩. Another restriction is that the continuation of each ⟨init⟩ should not be invoked more than once.

Implementation responsibilities: Implementations must, during the evaluation of an ⟨init⟩ expression, detect references to the value of the corresponding ⟨variable⟩ or the ⟨variable⟩ of any of the bindings that follow it in ⟨bindings⟩. If an implementation detects such a violation of the restriction, it must raise an exception with condition type `&assertion`. Implementations may or may not detect that the continuation of each ⟨init⟩ is invoked more than once. However, if the implementation detects this, it must raise an exception with condition type `&assertion`.

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

Syntax: ⟨Mv-bindings⟩ must have the form

(($\langle\text{formals}_1\rangle$ $\langle\text{init}_1\rangle$) ...),

where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 11.3. Any variable must not appear more than once in the set of ⟨formals⟩.

Semantics: The ⟨init⟩s are evaluated in the current environment (in some unspecified order), and the variables occurring in the ⟨formals⟩ are bound to fresh locations containing the values returned by the ⟨init⟩s, where the ⟨formals⟩ are matched to the return values in the same way that the ⟨formals⟩ in a `lambda` expression are matched to the arguments in a procedure call. Then, the ⟨body⟩ is evaluated in the extended environment, and the values of the last expression of ⟨body⟩ are returned. Each binding of a variable has ⟨body⟩ as its region. If the ⟨formals⟩ do not match, an exception with condition type `&assertion` is raised.

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

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

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

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

Syntax: ⟨Mv-bindings⟩ must have the form

(($\langle\text{formals}_1\rangle$ $\langle\text{init}_1\rangle$) ...), where each ⟨init⟩ is an expression, and ⟨body⟩ is as described in section 11.3. In each ⟨formals⟩, any variable must not appear more than once.

Semantics: The `let*-values` form is similar to `let-values`, but the ⟨init⟩s are evaluated and bindings created sequentially from left to right, with the region of the bindings of each ⟨formals⟩ including the bindings to its right as well as ⟨body⟩. Thus the second ⟨init⟩ is evaluated in an environment in which the bindings of the first ⟨formals⟩ is visible and initialized, and so on.

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

Note: While all of the variables bound by a `let-values` expression must be distinct, the variables bound by different ⟨formals⟩ of a `let*-values` expression need not be distinct.

11.4.7. Sequencing

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

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

- It may appear as a form in a ⟨body⟩ (see section 11.3), ⟨library body⟩ (see section 7.1), or ⟨top-level body⟩ (see chapter 8), or directly nested in a `begin` form that appears in a body. In this case, the `begin` form must have the shape specified in the first header line. This use of `begin` acts as a *splicing* form—the forms inside the ⟨body⟩ are spliced into the surrounding body, as if the `begin` wrapper were not actually present.

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

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

```
(define x 0)

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

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

11.5. Equivalence predicates

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

(`eqv?` obj_1 obj_2) procedure

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if obj_1 and obj_2 should normally be regarded as the same object and `#f` otherwise. This relation is left slightly open to interpretation, but the following partial specification of `eqv?` must hold for all implementations.

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

- Obj_1 and obj_2 are both booleans and are the same according to the `boolean=?` procedure (section 11.8).
- Obj_1 and obj_2 are both symbols and are the same according to the `symbol=?` procedure (section 11.10).
- Obj_1 and obj_2 are both exact number objects and are numerically equal (see `=`, section 11.7).
- Obj_1 and obj_2 are both inexact number objects, are numerically equal (see `=`, section 11.7), and yield the same results (in the sense of `eqv?`) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures, as long as the evaluation of that application does not involve NaN.
- Obj_1 and obj_2 are both characters and are the same character according to the `char=?` procedure (section 11.11).
- Both obj_1 and obj_2 are the empty list.
- Obj_1 and obj_2 are objects such as pairs, vectors, bytevectors (library chapter 2), strings, records (library chapter 6), ports (library section 8.2), or hashtables (library chapter 13) that refer to the same locations in the store (section 5.10).
- Obj_1 and obj_2 are record-type descriptors that are specified to be `eqv?` in library section 6.3.

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

- Obj_1 and obj_2 are of different types (section 11.1).
- Obj_1 and obj_2 are booleans for which the `boolean=?` procedure returns `#f`.

- *Obj*$_1$ and *obj*$_2$ are symbols for which the `symbol=?` procedure returns `#f`.
- One of *obj*$_1$ and *obj*$_2$ is an exact number object but the other is an inexact number object.
- *Obj*$_1$ and *obj*$_2$ are rational number objects for which the `=` procedure returns `#f`.
- *Obj*$_1$ and *obj*$_2$ yield different results (in the sense of `eqv?`) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures, as long as the evaluation of that application does not involve NaN.
- *Obj*$_1$ and *obj*$_2$ are characters for which the `char=?` procedure returns `#f`.
- One of *obj*$_1$ and *obj*$_2$ is the empty list, but the other is not.
- *Obj*$_1$ and *obj*$_2$ are objects such as pairs, vectors, bytevectors (library chapter 2), strings, records (library chapter 6), ports (library section 8.2), or hashtables (library chapter 13) that refer to distinct locations.
- *Obj*$_1$ and *obj*$_2$ are pairs, vectors, strings, or records, or hashtables, where the applying the same accessor (i.e. `car`, `cdr`, `vector-ref`, `string-ref`, or record accessors) to both yields results for which `eqv?` returns `#f`.
- *Obj*$_1$ and *obj*$_2$ are procedures that would behave differently (return different values or have different side effects) for some arguments.

Note: The `eqv?` procedure returning `#t` when *obj*$_1$ and *obj*$_2$ are number objects does not imply that `=` would also return `#t` when called with *obj*$_1$ and *obj*$_2$ as arguments.

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

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

```
(let ((p (lambda (x) x)))
  (eqv? p p))                 ⟹  unspecified
(eqv? "" "")                  ⟹  unspecified
(eqv? '#() '#())              ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))         ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))         ⟹  unspecified
(eqv? +nan.0 +nan.0)          ⟹  unspecified
```

The next set of examples shows the use of `eqv?` with procedures that have local state. Calls to `gen-counter` must return a distinct procedure every time, since each procedure has its own internal counter. Calls to `gen-loser` return procedures that behave equivalently when called. However, `eqv?` may not detect this equivalence.

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

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

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

Implementations may share structure between constants where appropriate. Thus the value of `eqv?` on constants is sometimes implementation-dependent.

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

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

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

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

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

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

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

The equal? predicate treats pairs and vectors as nodes with outgoing edges, uses string=? to compare strings, uses bytevector=? to compare bytevectors (see library chapter 2), and uses eqv? to compare other nodes.

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

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

Note: The equal? procedure must always terminate, even if its arguments contain cycles.

11.6. Procedure predicate

(procedure? *obj*) procedure

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

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

11.7. Arithmetic

The procedures described here implement arithmetic that is generic over the numerical tower described in chapter 3. The generic procedures described in this section accept both exact and inexact number objects as arguments, performing coercions and selecting the appropriate operations as determined by the numeric subtypes of their arguments.

Library chapter 11 describes libraries that define other numerical procedures.

11.7.1. Propagation of exactness and inexactness

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

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

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

```
/
div          mod           div-and-mod
div0         mod0          div0-and-mod0
```

Moreover, the procedure expt must return the correct exact result provided its first argument is an exact real number object and its second argument is an exact integer object.

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

One general exception to the rule above is that an implementation may return an exact result despite inexact arguments if that exact result would be the correct result for all possible substitutions of exact arguments for the inexact ones. An example is `(* 1.0 0)` which may return either `0` (exact) or `0.0` (inexact).

11.7.2. Representability of infinities and NaNs

The specification of the numerical operations is written as though infinities and NaNs are representable, and specifies many operations with respect to these number objects in ways that are consistent with the IEEE-754 standard for binary floating-point arithmetic. An implementation of Scheme may or may not represent infinities and NaNs; however, an implementation must raise a continuable exception with condition type `&no-infinities` or `&no-nans` (respectively; see library section 11.3) whenever it is unable to represent an infinity or NaN as specified. In this case, the continuation of the exception handler is the continuation that otherwise would have received the infinity or NaN value. This requirement also applies to conversions between number objects and external representations, including the reading of program source code.

11.7.3. Semantics of common operations

Some operations are the semantic basis for several arithmetic procedures. The behavior of these operations is described in this section for later reference.

Integer division

Scheme's operations for performing integer division rely on mathematical operations div, mod, div_0, and mod_0, that are defined as follows:

div, mod, div_0, and mod_0 each accept two real numbers x_1 and x_2 as operands, where x_2 must be nonzero.

div returns an integer, and mod returns a real. Their results are specified by

$$\begin{aligned} x_1 \text{ div } x_2 &= n_d \\ x_1 \text{ mod } x_2 &= x_m \end{aligned}$$

where

$$\begin{gathered} x_1 = n_d \cdot x_2 + x_m \\ 0 \le x_m < |x_2| \end{gathered}$$

Examples:

$$\begin{aligned} 123 \text{ div } 10 &= 12 \\ 123 \text{ mod } 10 &= 3 \\ 123 \text{ div } -10 &= -12 \\ 123 \text{ mod } -10 &= 3 \\ -123 \text{ div } 10 &= -13 \\ -123 \text{ mod } 10 &= 7 \\ -123 \text{ div } -10 &= 13 \\ -123 \text{ mod } -10 &= 7 \end{aligned}$$

div_0 and mod_0 are like div and mod, except the result of mod_0 lies within a half-open interval centered on zero. The results are specified by

$$\begin{aligned} x_1 \text{ div}_0\ x_2 &= n_d \\ x_1 \text{ mod}_0\ x_2 &= x_m \end{aligned}$$

where:

$$\begin{gathered} x_1 = n_d \cdot x_2 + x_m \\ -\left|\frac{x_2}{2}\right| \le x_m < \left|\frac{x_2}{2}\right| \end{gathered}$$

Examples:

$$\begin{aligned} 123 \text{ div}_0\ 10 &= 12 \\ 123 \text{ mod}_0\ 10 &= 3 \\ 123 \text{ div}_0\ -10 &= -12 \\ 123 \text{ mod}_0\ -10 &= 3 \\ -123 \text{ div}_0\ 10 &= -12 \\ -123 \text{ mod}_0\ 10 &= -3 \\ -123 \text{ div}_0\ -10 &= 12 \\ -123 \text{ mod}_0\ -10 &= -3 \end{aligned}$$

Transcendental functions

In general, the transcendental functions log, $\sin^{-1}$ (arcsine), $\cos^{-1}$ (arccosine), and $\tan^{-1}$ are multiply defined. The value of $\log z$ is defined to be the one whose imaginary part lies in the range from $-\pi$ (inclusive if -0.0 is distinguished, exclusive otherwise) to π (inclusive). $\log 0$ is undefined.

The value of $\log z$ for non-real z is defined in terms of log on real numbers as

$$\log z = \log |z| + (\text{angle } z)i$$

where angle z is the angle of $z = a \cdot e^{ib}$ specified as:

$$\text{angle } z = b + 2\pi n$$

with $-\pi \le \text{angle } z \le \pi$ and $\text{angle } z = b + 2\pi n$ for some integer n.

With the one-argument version of log defined this way, the values of the two-argument-version of log, $\sin^{-1} z$, $\cos^{-1} z$,

$\tan^{-1} z$, and the two-argument version of $\tan^{-1}$ are according to the following formulæ:

$$\begin{aligned}
\log z\ b &= \frac{\log z}{\log b} \\
\sin^{-1} z &= -i \log(iz + \sqrt{1 - z^2}) \\
\cos^{-1} z &= \pi/2 - \sin^{-1} z \\
\tan^{-1} z &= (\log(1 + iz) - \log(1 - iz))/(2i) \\
\tan^{-1} x\ y &= \text{angle}(x + yi)
\end{aligned}$$

The range of $\tan^{-1} x\ y$ is as in the following table. The asterisk (*) indicates that the entry applies to implementations that distinguish minus zero.

	y condition	x condition	range of result r
	$y = 0.0$	$x > 0.0$	0.0
$*$	$y = +0.0$	$x > 0.0$	$+0.0$
$*$	$y = -0.0$	$x > 0.0$	-0.0
	$y > 0.0$	$x > 0.0$	$0.0 < r < \frac{\pi}{2}$
	$y > 0.0$	$x = 0.0$	$\frac{\pi}{2}$
	$y > 0.0$	$x < 0.0$	$\frac{\pi}{2} < r < \pi$
	$y = 0.0$	$x < 0$	π
$*$	$y = +0.0$	$x < 0.0$	π
$*$	$y = -0.0$	$x < 0.0$	$-\pi$
	$y < 0.0$	$x < 0.0$	$-\pi < r < -\frac{\pi}{2}$
	$y < 0.0$	$x = 0.0$	$-\frac{\pi}{2}$
	$y < 0.0$	$x > 0.0$	$-\frac{\pi}{2} < r < 0.0$
	$y = 0.0$	$x = 0.0$	undefined
$*$	$y = +0.0$	$x = +0.0$	$+0.0$
$*$	$y = -0.0$	$x = +0.0$	-0.0
$*$	$y = +0.0$	$x = -0.0$	π
$*$	$y = -0.0$	$x = -0.0$	$-\pi$
$*$	$y = +0.0$	$x = 0$	$\frac{\pi}{2}$
$*$	$y = -0.0$	$x = 0$	$-\frac{\pi}{2}$

11.7.4. Numerical operations

Numerical type predicates

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

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

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

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

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

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

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

Note: Except for `number?`, the behavior of these type predicates on inexact number objects is unreliable, because any inaccuracy may affect the result.

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

These numerical type predicates can be applied to any kind of argument. The `real-valued?` procedure returns `#t` if the object is a number object and is equal in the sense of `=` to some real number object, or if the object is a NaN, or a complex number object whose real part is a NaN and whose imaginary part is zero in the sense of `zero?`. The `rational-valued?` and `integer-valued?` procedures return `#t` if the object is a number object and is equal in the sense of `=` to some object of the named type, and otherwise they return `#f`.

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

```
(real-valued? -2.5+0.0i)      ⟹  #t
(real-valued? -2.5+0i)        ⟹  #t
(real-valued? -2.5)           ⟹  #t
(real-valued? #e1e10)         ⟹  #t

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

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

Note: These procedures test whether a given number object can be coerced to the specified type without loss of numerical accuracy. Specifically, the behavior of these predicates differs from the behavior of `real?`, `rational?`, and `integer?` on complex number objects whose imaginary part is inexact zero.

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

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

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

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

Generic conversions

`(inexact` *z*`)` procedure
`(exact` *z*`)` procedure

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

Note: For a real number object whose magnitude is finite but so large that it has no reasonable finite approximation as an inexact number, a reasonably close inexact equivalent may be `+inf.0` or `-inf.0`. Similarly, the inexact representation of a complex number object whose components are finite may have infinite components.

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

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

The `inexact` and `exact` procedures are idempotent.

Arithmetic operations

`(=` z_1 z_2 z_3 `...)` procedure
`(<` x_1 x_2 x_3 `...)` procedure
`(>` x_1 x_2 x_3 `...)` procedure
`(<=` x_1 x_2 x_3 `...)` procedure
`(>=` x_1 x_2 x_3 `...)` procedure

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

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

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

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

For any number object z:

```
(= +nan.0 z)                  ⟹  #f
```

For any real number object x:

```
(< +nan.0 x)                  ⟹  #f
(> +nan.0 x)                  ⟹  #f
```

These predicates must be transitive.

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

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

When in doubt, consult a numerical analyst.

`(zero?` *z*`)` procedure
`(positive?` *x*`)` procedure
`(negative?` *x*`)` procedure
`(odd?` *n*`)` procedure

(even? n) procedure
(finite? x) procedure
(infinite? x) procedure
(nan? x) procedure

These numerical predicates test a number object for a particular property, returning `#t` or `#f`. The `zero?` procedure tests if the number object is `=` to zero, `positive?` tests whether it is greater than zero, `negative?` tests whether it is less than zero, `odd?` tests whether it is odd, `even?` tests whether it is even, `finite?` tests whether it is not an infinity and not a NaN, `infinite?` tests whether it is an infinity, `nan?` tests whether it is a NaN.

```
(zero? +0.0)          ⟹  #t
(zero? -0.0)          ⟹  #t
(zero? +nan.0)        ⟹  #f
(positive? +inf.0)    ⟹  #t
(negative? -inf.0)    ⟹  #t
(positive? +nan.0)    ⟹  #f
(negative? +nan.0)    ⟹  #f
(finite? +inf.0)      ⟹  #f
(finite? 5)           ⟹  #t
(finite? 5.0)         ⟹  #t
(infinite? 5.0)       ⟹  #f
(infinite? +inf.0)    ⟹  #t
```

Note: As with the predicates above, the results may be unreliable because a small inaccuracy may affect the result.

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

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)             ⟹  4
(max 3.9 4)           ⟹  4.0
```

For any real number object x that is not a NaN:

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

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

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

These procedures return the sum or product of their arguments.

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

(* 4)                 ⟹  4
(*)                   ⟹  1
(* 5 +inf.0)          ⟹  +inf.0
(* -5 +inf.0)         ⟹  -inf.0
(* +inf.0 +inf.0)     ⟹  +inf.0
(* +inf.0 -inf.0)     ⟹  -inf.0
(* 0 +inf.0)          ⟹  0 or +nan.0
(* 0 +nan.0)          ⟹  0 or +nan.0
(* 1.0 0)             ⟹  0 or 0.0
```

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

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

For any real number object x:

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

For any real number object x that is not an exact 0:

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

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

Implementations that distinguish -0.0 should adopt behavior consistent with the following examples:

```
(+ 0.0 -0.0)          ⟹  0.0
(+ -0.0 0.0)          ⟹  0.0
(+ 0.0 0.0)           ⟹  0.0
(+ -0.0 -0.0)         ⟹  -0.0
```

(- z) procedure
(- z_1 z_2 z_3 ...) procedure

With two or more arguments, this procedures returns the difference of its arguments, associating to the left. With one argument, however, it returns the additive inverse of its argument.

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

If this procedure is applied to mixed non-rational real and non-real complex arguments, it either raises an exception with condition type `&implementation-restriction` or returns an unspecified number object.

Implementations that distinguish -0.0 should adopt behavior consistent with the following examples:

```
(- 0.0)            ⟹ -0.0
(- -0.0)           ⟹ 0.0
(- 0.0 -0.0)       ⟹ 0.0
(- -0.0 0.0)       ⟹ -0.0
(- 0.0 0.0)        ⟹ 0.0
(- -0.0 -0.0)      ⟹ 0.0
```

(/ *z*) procedure
(/ z_1 z_2 z_3 ...) procedure

If all of the arguments are exact, then the divisors must all be nonzero. With two or more arguments, this procedure returns the quotient of its arguments, associating to the left. With one argument, however, it returns the multiplicative inverse of its argument.

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

If this procedure is applied to mixed non-rational real and non-real complex arguments, it either raises an exception with condition type `&implementation-restriction` or returns an unspecified number object.

(abs *x*) procedure

Returns the absolute value of its argument.

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

(div-and-mod x_1 x_2) procedure
(div x_1 x_2) procedure
(mod x_1 x_2) procedure
(div0-and-mod0 x_1 x_2) procedure
(div0 x_1 x_2) procedure
(mod0 x_1 x_2) procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in section 11.7.3. If x_1 and x_2 are exact, x_2 must be nonzero. In the cases where the mathematical requirements in section 11.7.3 cannot be satisfied by any number object, either an exception is raised with condition type `&implementation-restriction`, or unspecified number objects (one for `div` `mod`, `div0` and `mod0`, two for `div-and-mod` and `div0-and-mod0`) are returned.

(div x_1 x_2) ⟹ x_1 div x_2
(mod x_1 x_2) ⟹ x_1 mod x_2
(div-and-mod x_1 x_2) ⟹ x_1 div x_2, x_1 mod x_2
; two return values
(div0 x_1 x_2) ⟹ x_1 div_0 x_2
(mod0 x_1 x_2) ⟹ x_1 mod_0 x_2
(div0-and-mod0 x_1 x_2)
⟹ x_1 div_0 x_2, x_1 mod_0 x_2
; two return values

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

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

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

(numerator *q*) procedure
(denominator *q*) procedure

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

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

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

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

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

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

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

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

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

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

(rationalize x_1 x_2) procedure

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

```
(rationalize (exact .3) 1/10)
          ⟹ 1/3
(rationalize .3 1/10)
          ⟹ #i1/3  ; approximately

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

The first two examples hold only in implementations whose inexact real number objects have sufficient precision.

(exp z) procedure
(log z) procedure
(log z_1 z_2) procedure
(sin z) procedure
(cos z) procedure
(tan z) procedure
(asin z) procedure
(acos z) procedure
(atan z) procedure
(atan x_1 x_2) procedure

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

See section 11.7.3 for the underlying mathematical operations. These procedures may return inexact results even when given exact arguments.

```
(exp +inf.0)            ⟹ +inf.0
(exp -inf.0)            ⟹ 0.0
(log +inf.0)            ⟹ +inf.0
(log 0.0)               ⟹ -inf.0
(log 0)               ⟹ &assertion exception
(log -inf.0)
          ⟹ +inf.0+3.141592653589793i
           ; approximately
(atan -inf.0)
          ⟹ -1.5707963267948965 ; approximately
(atan +inf.0)
          ⟹ 1.5707963267948965 ; approximately
(log -1.0+0.0i)
          ⟹ 0.0+3.141592653589793i ; approximately
(log -1.0-0.0i)
          ⟹ 0.0-3.141592653589793i ; approximately
          ; if -0.0 is distinguished
```

(sqrt z) procedure

Returns the principal square root of z. For rational z, the result has either positive real part, or zero real part and non-negative imaginary part. With log defined as in section 11.7.3, the value of (sqrt z) could be expressed as $e^{\frac{\log z}{2}}$.

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

```
(sqrt -5)
          ⟹  0.0+2.23606797749979i ; approximately
(sqrt +inf.0)           ⟹  +inf.0
(sqrt -inf.0)           ⟹  +inf.0i
```

(exact-integer-sqrt k) procedure

The exact-integer-sqrt procedure returns two non-negative exact integer objects s and r where $k = s^2 + r$ and $k < (s + 1)^2$.

```
(exact-integer-sqrt 4)      ⟹ 2 0
                             ; two return values
(exact-integer-sqrt 5)      ⟹ 2 1
                             ; two return values
```

(expt z_1 z_2) procedure

Returns z_1 raised to the power z_2. For nonzero z_1, this is $e^{z_2 \log z_1}$. 0.0^z is 1.0 if $z = 0.0$, and 0.0 if (real-part z) is positive. For other cases in which the first argument is zero, either an exception is raised with condition type `&implementation-restriction`, or an unspecified number object is returned.

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

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

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

Suppose a_1, a_2, a_3, and a_4 are real numbers, and c is a complex number such that the following holds:

$$c = a_1 + a_2 i = a_3 e^{ia_4}$$

Then, if x_1, x_2, x_3, and x_4 are number objects representing a_1, a_2, a_3, and a_4, respectively, `(make-rectangular` x_1 x_2`)` returns c, and `(make-polar` x_3 x_4`)` returns c.

```
(make-rectangular 1.1 2.2)
          ⟹ 1.1+2.2i ; approximately
(make-polar 1.1 2.2)
          ⟹ 1.1@2.2 ; approximately
```

Conversely, if $-\pi \leq a_4 \leq \pi$, and if z is a number object representing c, then `(real-part` z`)` returns a_1 `(imag-part` z`)` returns a_2, `(magnitude` z`)` returns a_3, and `(angle` z`)` returns a_4.

```
(real-part 1.1+2.2i)       ⟹ 1.1 ; approximately
(imag-part 1.1+2.2i)       ⟹ 2.2 ; approximately
(magnitude 1.1@2.2)        ⟹ 1.1 ; approximately
(angle 1.1@2.2)            ⟹ 2.2 ; approximately

(angle -1.0)
          ⟹ 3.141592653589793 ; approximately
(angle -1.0+0.0i)
          ⟹ 3.141592653589793 ; approximately
(angle -1.0-0.0i)
          ⟹ -3.141592653589793 ; approximately
          ; if -0.0 is distinguished
(angle +inf.0)             ⟹ 0.0
(angle -inf.0)
          ⟹ 3.141592653589793 ; approximately
```

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

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

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

```
(angle -1)
          ⟹ 3.141592653589793 ; approximately
```

Numerical Input and Output

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

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

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

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

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

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

If z is inexact, the radix is 10, and the above expression and condition can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent, trailing zeroes, and mantissa width) needed to make the above expression and condition true [4, 7]; otherwise the format of the result is unspecified.

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

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

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

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

Note: If the arguments to `string->number` are a string and a valid radix as specified, it must produce a number object or `#f`; it may not raise an exception.

11.8. Booleans

The standard boolean objects for true and false have external representations `#t` and `#f`. However, of all objects, only `#f` counts as false in conditional expressions. See section 5.7.

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

(`not` *obj*) procedure

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

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

(`boolean?` *obj*) procedure

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

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

(`boolean=?` $bool_1$ $bool_2$ $bool_3$ ...) procedure

Returns `#t` if the booleans are the same.

11.9. Pairs and lists

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

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

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

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

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

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

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

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

is equivalent to

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

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

(`pair?` *obj*) procedure

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

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

(`cons` obj_1 obj_2) procedure

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

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

(car *pair*) procedure

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

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

(cdr *pair*) procedure

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

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

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

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

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

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

(null? *obj*) procedure

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

(list? *obj*) procedure

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

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

(list *obj* ...) procedure

Returns a newly allocated list of its arguments.

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

(length *list*) procedure

Returns the length of *list*.

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

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

Returns a possibly improper list consisting of the elements of the first *list* followed by the elements of the other *lists*, with *obj* as the cdr of the final pair. An improper list results if *obj* is not a list. The append procedure returns the empty list if called with no arguments.

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

The return value is made from new pairs for all arguments but the last; the last is merely placed at the end of the new structure.

(reverse *list*) procedure

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

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

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

List should be a list of size at least *k*. The list-tail procedure returns the subchain of pairs of *list* obtained by omitting the first *k* elements.

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

Implementation responsibilities: The implementation must check that *list* is a chain of pairs whose length is at least *k*. It should not check that it is a chain of pairs beyond this length.

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

List must be a list whose length is at least $k + 1$. The list-ref procedure returns the kth element of *list*.

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

Implementation responsibilities: The implementation must check that *list* is a chain of pairs whose length is at least $k + 1$. It should not check that it is a list of pairs beyond this length.

(`map` *proc* $list_1$ $list_2$ ...) procedure

The *lists* should all have the same length. *Proc* should accept as many arguments as there are *lists* and return a single value. *Proc* should not mutate any of the *lists*.

The `map` procedure applies *proc* element-wise to the elements of the *lists* and returns a list of the results, in order. *Proc* is always called in the same dynamic environment as `map` itself. The order in which *proc* is applied to the elements of the *lists* is unspecified. If multiple returns occur from `map`, the values returned by earlier returns are not mutated.

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

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

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

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

Implementation responsibilities: The implementation should check that the *lists* all have the same length. The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

(`for-each` *proc* $list_1$ $list_2$ ...) procedure

The *lists* should all have the same length. *Proc* should accept as many arguments as there are *lists*. *Proc* should not mutate any of the *lists*.

The `for-each` procedure applies *proc* element-wise to the elements of the *lists* for its side effects, in order from the first elements to the last. *Proc* is always called in the same dynamic environment as `for-each` itself. The return values of `for-each` are unspecified.

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

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

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

Implementation responsibilities: The implementation should check that the *lists* all have the same length. The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

Note: Implementations of `for-each` may or may not tail-call *proc* on the last elements.

11.10. Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eq?`, `eqv?` and `equal?`) if and only if their names are spelled the same way. A symbol literal is formed using `quote`.

(`symbol?` *obj*) procedure

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

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

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

Returns the name of *symbol* as an immutable string.

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

(`symbol=?` $symbol_1$ $symbol_2$ $symbol_3$ ...) procedure

Returns `#t` if the symbols are the same, i.e., if their names are spelled the same.

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

Returns the symbol whose name is *string*.

```
(eq? 'mISSISSIppi 'mississippi)
        ⟹  #f
(string->symbol "mISSISSIppi")
        ⟹  the symbol with name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
        ⟹  #t
(eq? 'JollyWog
```

```
        (string->symbol
          (symbol->string 'JollyWog)))
             ⟹  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
             ⟹  #t
```

11.11. Characters

Characters are objects that represent Unicode scalar values [27].

Note: Unicode defines a standard mapping between sequences of *Unicode scalar values* (integers in the range 0 to #x10FFFF, excluding the range #xD800 to #xDFFF) in the latest version of the standard and human-readable "characters". More precisely, Unicode distinguishes between glyphs, which are printed for humans to read, and characters, which are abstract entities that map to glyphs (sometimes in a way that's sensitive to surrounding characters). Furthermore, different sequences of scalar values sometimes correspond to the same character. The relationships among scalar, characters, and glyphs are subtle and complex.

Despite this complexity, most things that a literate human would call a "character" can be represented by a single Unicode scalar value (although several sequences of Unicode scalar values may represent that same character). For example, Roman letters, Cyrillic letters, Hebrew consonants, and most Chinese characters fall into this category.

Unicode scalar values exclude the range #xD800 to #xDFFF, which are part of the range of Unicode *code points*. However, the Unicode code points in this range, the so-called *surrogates*, are an artifact of the UTF-16 encoding, and can only appear in specific Unicode encodings, and even then only in pairs that encode scalar values. Consequently, all characters represent code points, but the surrogate code points do not have representations as characters.

(`char?` *obj*) procedure

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

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

Sv must be a Unicode scalar value, i.e., a non-negative exact integer object in $[0, \#x\text{D7FF}] \cup [\#x\text{E000}, \#x10\text{FFFF}]$.

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

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

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

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

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

11.12. Strings

Strings are sequences of characters.

The *length* of a string is the number of characters that it contains. This number is fixed when the string is created. The *valid indices* of a string are the integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

(`string?` *obj*) procedure

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

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

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

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

Returns a newly allocated string composed of the arguments.

(`string-length` *string*) procedure

Returns the number of characters in the given *string* as an exact integer object.

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

K must be a valid index of *string*. The `string-ref` procedure returns character *k* of *string* using zero-origin indexing.

Note: Implementors should make `string-ref` run in constant time.

(`string=?` $string_1$ $string_2$ $string_3$ ...) procedure

Returns `#t` if the strings are the same length and contain the same characters in the same positions. Otherwise, the `string=?` procedure returns `#f`.

```
(string=? "Straße" "Strasse")
        ⟹ #f
```

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

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

```
(string<? "z" "ß")        ⟹ #t
(string<? "z" "zz")       ⟹ #t
(string<? "z" "Z")        ⟹ #f
```

`(substring string start end)` procedure

String must be a string, and *start* and *end* must be exact integer objects satisfying

$$0 \leq start \leq end \leq \texttt{(string-length } string\texttt{)}.$$

The `substring` procedure returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

`(string-append string ...)` procedure

Returns a newly allocated string whose characters form the concatenation of the given strings.

`(string->list string)` procedure
`(list->string list)` procedure

List must be a list of characters. The `string->list` procedure returns a newly allocated list of the characters that make up the given string. The `list->string` procedure returns a newly allocated string formed from the characters in *list*. The `string->list` and `list->string` procedures are inverses so far as `equal?` is concerned.

`(string-for-each proc string1 string2 ...)` procedure

The *strings* must all have the same length. *Proc* should accept as many arguments as there are *strings*. The `string-for-each` procedure applies *proc* element-wise to the characters of the *strings* for its side effects, in order from the first characters to the last. *Proc* is always called in the same dynamic environment as `string-for-each` itself. The return values of `string-for-each` are unspecified.

Analogous to `for-each`.

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

`(string-copy string)` procedure

Returns a newly allocated copy of the given *string*.

11.13. Vectors

Vectors are heterogeneous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time needed to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indices* of a vector are the exact non-negative integer objects less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Like list constants, vector constants must be quoted:

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

`(vector? obj)` procedure

Returns `#t` if *obj* is a vector. Otherwise the procedure returns `#f`.

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

Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

`(vector obj ...)` procedure

Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)        ⟹  #(a b c)
```

`(vector-length vector)` procedure

Returns the number of elements in *vector* as an exact integer object.

`(vector-ref vector k)` procedure

K must be a valid index of *vector*. The `vector-ref` procedure returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)
          ⟹  8
```

`(vector-set! vector k obj)` procedure

K must be a valid index of *vector*. The `vector-set!` procedure stores *obj* in element *k* of *vector*, and returns unspecified values.

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

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
          ⟹  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
          ⟹  unspecified
             ; constant vector
             ; should raise &assertion exception
```

`(vector->list vector)` procedure
`(list->vector list)` procedure

The `vector->list` procedure returns a newly allocated list of the objects contained in the elements of *vector*. The `list->vector` procedure returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '#(dah dah didah))
          ⟹  (dah dah didah)
(list->vector '(dididit dah))
          ⟹  #(dididit dah)
```

`(vector-fill! vector fill)` procedure

Stores *fill* in every element of *vector* and returns unspecified values.

`(vector-map proc vector`$_1$ `vector`$_2$ `...)` procedure

The *vector*s must all have the same length. *Proc* should accept as many arguments as there are *vector*s and return a single value.

The `vector-map` procedure applies *proc* element-wise to the elements of the *vector*s and returns a vector of the results, in order. *Proc* is always called in the same dynamic environment as `vector-map` itself. The order in which *proc* is applied to the elements of the *vector*s is unspecified. If multiple returns occur from `vector-map`, the return values returned by earlier returns are not mutated.

Analogous to `map`.

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

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

The *vector*s must all have the same length. *Proc* should accept as many arguments as there are *vector*s. The `vector-for-each` procedure applies *proc* element-wise to the elements of the *vector*s for its side effects, in order from the first elements to the last. *Proc* is always called in the same dynamic environment as `vector-for-each` itself. The return values of `vector-for-each` are unspecified.

Analogous to `for-each`.

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

11.14. Errors and violations

`(error who message irritant`$_1$ `...)` procedure
`(assertion-violation who message irritant`$_1$ `...)` procedure

Who must be a string or a symbol or `#f`. *Message* must be a string. The *irritant*s are arbitrary objects.

These procedures raise an exception. The `error` procedure should be called when an error has occurred, typically caused by something that has gone wrong in the interaction of the program with the external world or the user. The `assertion-violation` procedure should be called when an invalid call to a procedure was made, either passing an invalid number of arguments, or passing an argument that it is not specified to handle.

The *who* argument should describe the procedure or operation that detected the exception. The *message* argument should describe the exceptional situation. The *irritant*s should be the arguments to the operation that detected the operation.

The condition object provided with the exception (see library chapter 7) has the following condition types:

- If *who* is not `#f`, the condition has condition type `&who`, with *who* as the value of its field. In that case, *who* should be the name of the procedure or entity that detected the exception. If it is `#f`, the condition does not have condition type `&who`.
- The condition has condition type `&message`, with *message* as the value of its field.

- The condition has condition type `&irritants`, and its field has as its value a list of the *irritant*s.

Moreover, the condition created by `error` has condition type `&error`, and the condition created by `assertion-violation` has condition type `&assertion`.

```
(define (fac n)
  (if (not (integer-valued? n))
      (assertion-violation
       'fac "non-integral argument" n))
  (if (negative? n)
      (assertion-violation
       'fac "negative argument" n))
  (letrec
    ((loop (lambda (n r)
             (if (zero? n)
                 r
                 (loop (- n 1) (* r n))))))
      (loop n 1)))

(fac 5)                    ⟹ 120
(fac 4.5)                ⟹ &assertion exception
(fac -3)                 ⟹ &assertion exception
```

`(assert` ⟨expression⟩`)` syntax

An `assert` form is evaluated by evaluating ⟨expression⟩. If ⟨expression⟩ returns a true value, that value is returned from the `assert` expression. If ⟨expression⟩ returns `#f`, an exception with condition types `&assertion` and `&message` is raised. The message provided in the condition object is implementation-dependent.

Note: Implementations should exploit the fact that `assert` is syntax to provide as much information as possible about the location of the assertion failure.

11.15. Control features

This chapter describes various primitive procedures which control the flow of program execution in special ways.

`(apply` *proc* arg_1 … *rest-args*`)` procedure

Rest-args must be a list. *Proc* should accept n arguments, where n is number of *args* plus the length of *rest-args*. The `apply` procedure calls *proc* with the elements of the list `(append (list` arg_1 `...)` *rest-args*`)` as the actual arguments.

If a call to `apply` occurs in a tail context, the call to *proc* is also in a tail context.

```
(apply + (list 3 4))          ⟹  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)     ⟹  30
```

`(call-with-current-continuation` *proc*`)` procedure
`(call/cc` *proc*`)` procedure

Proc should accept one argument. The procedure `call-with-current-continuation` (which is the same as the procedure `call/cc`) packages the current continuation as an "escape procedure" and passes it as an argument to *proc*. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead reinstate the continuation that was in effect when the escape procedure was created. Calling the escape procedure may cause the invocation of *before* and *after* procedures installed using `dynamic-wind`.

The escape procedure accepts the same number of arguments as the continuation of the original call to `call-with-current-continuation`.

The escape procedure that is passed to *proc* has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

If a call to `call-with-current-continuation` occurs in a tail context, the call to *proc* is also in a tail context.

The following examples show only some ways in which `call-with-current-continuation` is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of `call-with-current-continuation`.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                       ⟹  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))      ⟹  4
```

```
(list-length '(a b . c))     ⟹  #f
(call-with-current-continuation procedure?)
                              ⟹  #t
```

Note: Calling an escape procedure reenters the dynamic extent of the call to `call-with-current-continuation`, and thus restores its dynamic environment; see section 5.12.

`(values` *obj* ...`)` procedure

Delivers all of its arguments to its continuation. The `values` procedure might be defined as follows:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

The continuations of all non-final expressions within a sequence of expressions, such as in `lambda`, `begin`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `case`, and `cond` forms, usually take an arbitrary number of values.

Except for these and the continuations created by `call-with-values`, `let-values`, and `let*-values`, continuations implicitly accepting a single value, such as the continuations of ⟨operator⟩ and ⟨operand⟩s of procedure calls or the ⟨test⟩ expressions in conditionals, take exactly one value. The effect of passing an inappropriate number of values to such a continuation is undefined.

`(call-with-values` *producer consumer*`)` procedure

Producer must be a procedure and should accept zero arguments. *Consumer* must be a procedure and should accept as many values as *producer* returns. The `call-with-values` procedure calls *producer* with no arguments and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                              ⟹  5

(call-with-values * -)        ⟹  -1
```

If a call to `call-with-values` occurs in a tail context, the call to *consumer* is also in a tail context.

Implementation responsibilities: After *producer* returns, the implementation must check that *consumer* accepts as many values as *consumer* has returned.

`(dynamic-wind` *before thunk after*`)` procedure

Before, *thunk*, and *after* must be procedures, and each should accept zero arguments. These procedures may return any number of values. The `dynamic-wind` procedure calls *thunk* without arguments, returning the results of this call. Moreover, `dynamic-wind` calls *before* without arguments whenever the dynamic extent of the call to *thunk* is entered, and *after* without arguments whenever the dynamic extent of the call to *thunk* is exited. Thus, in the absence of calls to escape procedures created by `call-with-current-continuation`, `dynamic-wind` calls *before*, *thunk*, and *after*, in that order.

While the calls to *before* and *after* are not considered to be within the dynamic extent of the call to *thunk*, calls to the *before* and *after* procedures of any other calls to `dynamic-wind` that occur within the dynamic extent of the call to *thunk* are considered to be within the dynamic extent of the call to *thunk*.

More precisely, an escape procedure transfers control out of the dynamic extent of a set of zero or more active `dynamic-wind` calls x ... and transfer control into the dynamic extent of a set of zero or more active `dynamic-wind` calls y .... It leaves the dynamic extent of the most recent x and calls without arguments the corresponding *after* procedure. If the *after* procedure returns, the escape procedure proceeds to the next most recent x, and so on. Once each x has been handled in this manner, the escape procedure calls without arguments the *before* procedure corresponding to the least recent y. If the *before* procedure returns, the escape procedure reenters the dynamic extent of the least recent y and proceeds with the next least recent y, and so on. Once each y has been handled in this manner, control is transferred to the continuation packaged in the escape procedure.

Implementation responsibilities: The implementation must check the restrictions on *thunk* and *after* only if they are actually called.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

          ⟹ (connect talk1 disconnect
              connect talk2 disconnect)

(let ((n 0))
  (call-with-current-continuation
    (lambda (k)
      (dynamic-wind
```

```
          (lambda ()
            (set! n (+ n 1))
            (k))
          (lambda ()
            (set! n (+ n 2)))
          (lambda ()
            (set! n (+ n 4))))))
    n)                          ⟹ 1

(let ((n 0))
  (call-with-current-continuation
    (lambda (k)
      (dynamic-wind
        values
        (lambda ()
          (dynamic-wind
            values
            (lambda ()
              (set! n (+ n 1))
              (k))
            (lambda ()
              (set! n (+ n 2))
              (k))))
        (lambda ()
          (set! n (+ n 4))))))
  n)                            ⟹ 7
```

Note: Entering a dynamic extent restores its dynamic environment; see section 5.12.

11.16. Iteration

(`let` ⟨variable⟩ ⟨bindings⟩ ⟨body⟩) syntax

"Named `let`" is a variant on the syntax of `let` that provides a general looping construct and may also be used to express recursion. It has the same syntax and semantics as ordinary `let` except that ⟨variable⟩ is bound within ⟨body⟩ to a procedure whose parameters are the bound variables and whose body is ⟨body⟩. Thus the execution of ⟨body⟩ may be repeated by invoking the procedure named by ⟨variable⟩.

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))
     ⟹  ((6 1 3) (-5 -2))
```

11.17. Quasiquotation

(`quasiquote` ⟨qq template⟩) syntax
`unquote` auxiliary syntax
`unquote-splicing` auxiliary syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance.

Syntax: ⟨Qq template⟩ should be as specified by the grammar at the end of this entry.

Semantics: If no `unquote` or `unquote-splicing` forms appear within the ⟨qq template⟩, the result of evaluating (`quasiquote` ⟨qq template⟩) is equivalent to the result of evaluating (`quote` ⟨qq template⟩).

If an (`unquote` ⟨expression⟩ ...) form appears inside a ⟨qq template⟩, however, the ⟨expression⟩s are evaluated ("unquoted") and their results are inserted into the structure instead of the `unquote` form.

If an (`unquote-splicing` ⟨expression⟩ ...) form appears inside a ⟨qq template⟩, then the ⟨expression⟩s must evaluate to lists; the opening and closing parentheses of the lists are then "stripped away" and the elements of the lists are inserted in place of the `unquote-splicing` form.

Any `unquote-splicing` or multi-operand `unquote` form must appear only within a list or vector ⟨qq template⟩.

As noted in section 4.3.5, (`quasiquote` ⟨qq template⟩) may be abbreviated `` ` ``⟨qq template⟩, (`unquote` ⟨expression⟩) may be abbreviated `,`⟨expression⟩, and (`unquote-splicing` ⟨expression⟩) may be abbreviated `,@`⟨expression⟩.

```
`(list ,(+ 1 2) 4)          ⟹  (list 3 4)
(let ((name 'a)) `(list ,name ',name))
          ⟹  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
          ⟹  (a 3 4 5 6 b)
`(( foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
          ⟹  ((foo 7) . cons)
`#(10 5 ,(- 4) ,@(map - '(16 9)) 8)
          ⟹  #(10 5 -4 -16 -9 8)
(let ((name 'foo))
  `((unquote name name name)))
          ⟹ (foo foo foo)
(let ((name '(foo)))
  `((unquote-splicing name name name)))
          ⟹ (foo foo foo)
(let ((q '((append x y) (sqrt 9))))
  ``(foo ,,@q))
          ⟹ `(foo
                (unquote (append x y) (sqrt 9)))
(let ((x '(2 3))
      (y '(4 5)))
  `(foo (unquote (append x y) (- 9))))
          ⟹ (foo (2 3 4 5) -9)
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost `quasiquote`. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ⟹  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
          ⟹  (a `(b ,x ,'y d) e)
```

A `quasiquote` expression may return either fresh, mutable objects or literal structure for any structure that is constructed at run time during the evaluation of the expression. Portions that do not need to be rebuilt are always literal. Thus,

```
(let ((a 3)) `((1 2) ,a ,4 ,'five 6))
```

may be equivalent to either of the following expressions:

```
'((1 2) 3 4 five 6)
(let ((a 3))
  (cons '(1 2)
        (cons a (cons 4 (cons 'five '(6))))))
```

However, it is not equivalent to this expression:

```
(let ((a 3)) (list (list 1 2) a 4 'five 6))
```

It is a syntax violation if any of the identifiers `quasiquote`, `unquote`, or `unquote-splicing` appear in positions within a ⟨qq template⟩ otherwise than as described above.

The following grammar for quasiquote expressions is not context-free. It is presented as a recipe for generating an infinite number of production rules. Imagine a copy of the following rules for $D = 1, 2, 3, \ldots$. D keeps track of the nesting depth.

⟨qq template⟩ ⟶ ⟨qq template 1⟩
⟨qq template 0⟩ ⟶ ⟨expression⟩
⟨quasiquotation D⟩ ⟶ (`quasiquote` ⟨qq template D⟩)
⟨qq template D⟩ ⟶ ⟨lexeme datum⟩
  | ⟨list qq template D⟩
  | ⟨vector qq template D⟩
  | ⟨unquotation D⟩
⟨list qq template D⟩ ⟶ (⟨qq template or splice D⟩*)
  | (⟨qq template or splice D⟩$^+$. ⟨qq template D⟩)
  | ⟨quasiquotation $D + 1$⟩
⟨vector qq template D⟩ ⟶ #(⟨qq template or splice D⟩*)
⟨unquotation D⟩ ⟶ (`unquote` ⟨qq template $D - 1$⟩)
⟨qq template or splice D⟩ ⟶ ⟨qq template D⟩
  | ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶
    (`unquote-splicing` ⟨qq template $D - 1$⟩*)
  | (`unquote` ⟨qq template $D - 1$⟩*)

In ⟨quasiquotation⟩s, a ⟨list qq template D⟩ can sometimes be confused with either an ⟨unquotation D⟩ or a ⟨splicing unquotation D⟩. The interpretation as an ⟨unquotation⟩ or ⟨splicing unquotation D⟩ takes precedence.

11.18. Binding constructs for syntactic keywords

The `let-syntax` and `letrec-syntax` forms bind keywords. Like a `begin` form, a `let-syntax` or `letrec-syntax` form may appear in a definition context, in which case it is treated as a definition, and the forms in the body must also be definitions. A `let-syntax` or `letrec-syntax` form may also appear in an expression context, in which case the forms within their bodies must be expressions.

(`let-syntax` ⟨bindings⟩ ⟨form⟩ ...) syntax

Syntax: ⟨Bindings⟩ must have the form

((⟨keyword⟩ ⟨expression⟩) ...)

Each ⟨keyword⟩ is an identifier, and each ⟨expression⟩ is an expression that evaluates, at macro-expansion time, to a *transformer*. Transformers may be created by `syntax-rules` or `identifier-syntax` (see section 11.19) or by one of the other mechanisms described in library chapter 12. It is a syntax violation for ⟨keyword⟩ to appear more than once in the list of keywords being bound.

Semantics: The ⟨form⟩s are expanded in the syntactic environment obtained by extending the syntactic environment of the `let-syntax` form with macros whose keywords are the ⟨keyword⟩s, bound to the specified transformers. Each binding of a ⟨keyword⟩ has the ⟨form⟩s as its region.

The ⟨form⟩s of a `let-syntax` form are treated, whether in definition or expression context, as if wrapped in an implicit `begin`; see section 11.4.7. Thus definitions in the result of expanding the ⟨form⟩s have the same region as any definition appearing in place of the `let-syntax` form would have.

Implementation responsibilities: The implementation should detect if the value of ⟨expression⟩ cannot possibly be a transformer.

```
(let-syntax ((when (syntax-rules ()
                     ((when test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now))
    if))                    ⟹  now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
```

```
    (let ((x 'inner))
      (m))))              ⟹  outer
(let ()
  (let-syntax
    ((def (syntax-rules ()
            ((def stuff ...) (define stuff ...)))))
    (def foo 42))
  foo)                    ⟹ 42

(let ()
  (let-syntax ())
  5)                      ⟹ 5
```

(letrec-syntax ⟨bindings⟩ ⟨form⟩ ...) syntax

Syntax: Same as for `let-syntax`.

Semantics: The ⟨form⟩s are expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` form with macros whose keywords are the ⟨keyword⟩s, bound to the specified transformers. Each binding of a ⟨keyword⟩ has the ⟨bindings⟩ as well as the ⟨form⟩s within its region, so the transformers can transcribe forms into uses of the macros introduced by the `letrec-syntax` form.

The ⟨form⟩s of a `letrec-syntax` form are treated, whether in definition or expression context, as if wrapped in an implicit `begin`; see section 11.4.7. Thus definitions in the result of expanding the ⟨form⟩s have the same region as any definition appearing in place of the `letrec-syntax` form would have.

Implementation responsibilities: The implementation should detect if the value of ⟨expression⟩ cannot possibly be a transformer.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))            ⟹  7
```

The following example highlights how `let-syntax` and `letrec-syntax` differ.

```
(let ((f (lambda (x) (+ x 1))))
  (let-syntax ((f (syntax-rules ()
                    ((f x) x)))
               (g (syntax-rules ()
                    ((g x) (f x)))))
    (list (f 1) (g 1))))
          ⟹ (1 2)

(let ((f (lambda (x) (+ x 1))))
  (letrec-syntax ((f (syntax-rules ()
                       ((f x) x)))
                  (g (syntax-rules ()
                       ((g x) (f x)))))
    (list (f 1) (g 1))))
          ⟹ (1 1)
```

The two expressions are identical except that the `let-syntax` form in the first expression is a `letrec-syntax` form in the second. In the first expression, the `f` occurring in `g` refers to the `let`-bound variable `f`, whereas in the second it refers to the keyword `f` whose binding is established by the `letrec-syntax` form.

11.19. Macro transformers

(syntax-rules (⟨literal⟩ ...) ⟨syntax rule⟩ ...) syntax (expand)

_ auxiliary syntax (expand)

... auxiliary syntax (expand)

Syntax: Each ⟨literal⟩ must be an identifier. Each ⟨syntax rule⟩ must have the following form:

(⟨srpattern⟩ ⟨template⟩)

An ⟨srpattern⟩ is a restricted form of ⟨pattern⟩, namely, a nonempty ⟨pattern⟩ in one of four parenthesized forms below whose first subform is an identifier or an underscore _. A ⟨pattern⟩ is an identifier, constant, or one of the following.

(⟨pattern⟩ ...)
(⟨pattern⟩ ⟨pattern⟩ ... . ⟨pattern⟩)
(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ...)
(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ... . ⟨pattern⟩)
#(⟨pattern⟩ ...)
#(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ...)

An ⟨ellipsis⟩ is the identifier "..." (three periods).

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

(⟨subtemplate⟩ ...)
(⟨subtemplate⟩ ... . ⟨template⟩)
#(⟨subtemplate⟩ ...)

A ⟨subtemplate⟩ is a ⟨template⟩ followed by zero or more ellipses.

Semantics: An instance of `syntax-rules` evaluates, at macro-expansion time, to a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the ⟨syntax rule⟩s, beginning with the leftmost ⟨syntax rule⟩. When a match is found, the macro use is transcribed hygienically according to the template. It is a syntax violation when no match is found.

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

While the first subform of ⟨srpattern⟩ may be an identifier, the identifier is not involved in the matching and is not considered a pattern variable or literal identifier.

Pattern variables match arbitrary input subforms and are used to refer to elements of the input. It is a syntax violation if the same pattern variable appears more than once in a ⟨pattern⟩.

Underscores also match arbitrary input subforms but are not pattern variables and so cannot be used to refer to those elements. Multiple underscores may appear in a ⟨pattern⟩.

A literal identifier matches an input subform if and only if the input subform is an identifier and either both its occurrence in the input expression and its occurrence in the list of literals have the same lexical binding, or the two identifiers have the same name and both have no lexical binding.

A subpattern followed by an ellipsis can match zero or more elements of the input.

More formally, an input form F matches a pattern P if and only if one of the following holds:

- P is an underscore (_).
- P is a pattern variable.
- P is a literal identifier and F is an identifier such that both P and F would refer to the same binding if both were to appear in the output of the macro outside of any bindings inserted into the output of the macro. (If neither of two like-named identifiers refers to any binding, i.e., both are undefined, they are considered to refer to the same binding.)
- P is of the form (P_1 ... P_n) and F is a list of n elements that match P_1 through P_n.
- P is of the form (P_1 ... P_n . P_x) and F is a list or improper list of n or more elements whose first n elements match P_1 through P_n and whose nth cdr matches P_x.
- P is of the form (P_1 ... P_k P_e ⟨ellipsis⟩ P_{m+1} ... P_n), where ⟨ellipsis⟩ is the identifier ... and F is a list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.
- P is of the form (P_1 ... P_k P_e ⟨ellipsis⟩ P_{m+1} ... P_n . P_x), where ⟨ellipsis⟩ is the identifier ... and F is a list or improper list of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, whose next $n-m$ elements match P_{m+1} through P_n, and whose nth and final cdr matches P_x.
- P is of the form #(P_1 ... P_n) and F is a vector of n elements that match P_1 through P_n.
- P is of the form #(P_1 ... P_k P_e ⟨ellipsis⟩ P_{m+1} ... P_n), where ⟨ellipsis⟩ is the identifier ... and F is a vector of n elements whose first k elements match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n.
- P is a pattern datum (any nonlist, nonvector, nonsymbol datum) and F is equal to P in the sense of the `equal?` procedure.

When a macro use is transcribed according to the template of the matching ⟨syntax rule⟩, pattern variables that occur in the template are replaced by the subforms they match in the input.

Pattern data and identifiers that are not pattern variables or ellipses are copied into the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the subtemplate than in the associated subpattern, the input form is replicated as necessary. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis, and for at least one such pattern variable, the subtemplate must be followed by exactly as many ellipses as the subpattern in which the pattern variable appears. (Otherwise, the expander would not be able to determine how many times the subform should be repeated in the output.) It is a syntax violation if the constraints of this paragraph are not met.

A template of the form (⟨ellipsis⟩ ⟨template⟩) is identical to ⟨template⟩, except that ellipses within the template have no special meaning. That is, any ellipses contained within ⟨template⟩ are treated as ordinary identifiers. In particular, the template `(... ...)` produces a single ellipsis, `...`. This allows syntactic abstractions to expand into forms containing ellipses.

```
(define-syntax be-like-begin
  (syntax-rules ()
    ((be-like-begin name)
     (define-syntax name
       (syntax-rules ()
         ((name expr (... ...))
          (begin expr (... ...))))))))

(be-like-begin sequence)
(sequence 1 2 3 4)          ⟹ 4
```

As an example for hygienic use of auxiliary identifier, if `let` and `cond` are defined as in section 11.4.6 and appendix B then they are hygienic (as required) and the following is not an error.

```
(let ((=> #f))
  (cond (#t => 'ok)))       ⟹ ok
```

The macro transformer for `cond` recognizes `=>` as a local variable, and hence an expression, and not as the identifier `=>`, which the macro transformer treats as a syntactic keyword. Thus the example expands into

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

instead of

```
(let ((=> #f))
  (let ((temp #t))
    (if temp ('ok temp))))
```

which would result in an assertion violation.

`(identifier-syntax ⟨template⟩)` syntax (`expand`)
`(identifier-syntax` syntax (`expand`)
  `(⟨id₁⟩ ⟨template₁⟩)`
  `((set! ⟨id₂⟩ ⟨pattern⟩)`
   `⟨template₂⟩))`
`set!` auxiliary syntax (`expand`)

Syntax: The ⟨id⟩s must be identifiers. The ⟨template⟩s must be as for `syntax-rules`.

Semantics: When a keyword is bound to a transformer produced by the first form of `identifier-syntax`, references to the keyword within the scope of the binding are replaced by ⟨template⟩.

```
(define p (cons 4 5))
(define-syntax p.car (identifier-syntax (car p)))
p.car                       ⟹ 4
(set! p.car 15)             ⟹ &syntax exception
```

The second, more general, form of `identifier-syntax` permits the transformer to determine what happens when `set!` is used. In this case, uses of the identifier by itself are replaced by ⟨template₁⟩, and uses of `set!` with the identifier are replaced by ⟨template₂⟩.

```
(define p (cons 4 5))
(define-syntax p.car
  (identifier-syntax
    (_ (car p))
    ((set! _ e) (set-car! p e))))
(set! p.car 15)
p.car                       ⟹ 15
p                           ⟹ (15 . 5)
```

11.20. Tail calls and tail contexts

A *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined inductively. Note that a tail context is always determined with respect to a particular lambda expression.

- The last expression within the body of a lambda expression, shown as ⟨tail expression⟩ below, occurs in a tail context.

```
(lambda ⟨formals⟩
  ⟨definition⟩*
  ⟨expression⟩* ⟨tail expression⟩)
```

- If one of the following expressions is in a tail context, then the subexpressions shown as ⟨tail expression⟩ are in a tail context. These were derived from specifications of the syntax of the forms described in this chapter by replacing some occurrences of ⟨expression⟩ with ⟨tail expression⟩. Only those rules that contain tail contexts are shown here.

```
(if ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩)
(if ⟨expression⟩ ⟨tail expression⟩)

(cond ⟨cond clause⟩+)
(cond ⟨cond clause⟩* (else ⟨tail sequence⟩))

(case ⟨expression⟩
  ⟨case clause⟩+)
(case ⟨expression⟩
  ⟨case clause⟩*
  (else ⟨tail sequence⟩))

(and ⟨expression⟩* ⟨tail expression⟩)
(or ⟨expression⟩* ⟨tail expression⟩)

(let ⟨bindings⟩ ⟨tail body⟩)
(let ⟨variable⟩ ⟨bindings⟩ ⟨tail body⟩)
(let* ⟨bindings⟩ ⟨tail body⟩)
(letrec* ⟨bindings⟩ ⟨tail body⟩)
(letrec ⟨bindings⟩ ⟨tail body⟩)
```

(let-values ⟨mv-bindings⟩ ⟨tail body⟩)
(let*-values ⟨mv-bindings⟩ ⟨tail body⟩)

(let-syntax ⟨bindings⟩ ⟨tail body⟩)
(letrec-syntax ⟨bindings⟩ ⟨tail body⟩)

(begin ⟨tail sequence⟩)

A ⟨cond clause⟩ is

(⟨test⟩ ⟨tail sequence⟩),

a ⟨case clause⟩ is

((⟨datum⟩*) ⟨tail sequence⟩),

a ⟨tail body⟩ is

⟨definition⟩* ⟨tail sequence⟩,

and a ⟨tail sequence⟩ is

⟨expression⟩* ⟨tail expression⟩.

- If a `cond` expression is in a tail context, and has a clause of the form (⟨expression$_1$⟩ => ⟨expression$_2$⟩) then the (implied) call to the procedure that results from the evaluation of ⟨expression$_2$⟩ is in a tail context. ⟨Expression$_2$⟩ itself is not in a tail context.

Certain built-in procedures must also perform tail calls. The first argument passed to `apply` and to `call-with-current-continuation`, and the second argument passed to `call-with-values`, must be called via a tail call.

In the following example the only tail call is the call to `f`. None of the calls to `g` or `h` are tail calls. The reference to `x` is in a tail context, but it is not a call and thus is not a tail call.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

Note: Implementations may recognize that some non-tail calls, such as the call to `h` above, can be evaluated as though they were tail calls. In the example above, the `let` expression could be compiled as a tail call to `h`. (The possibility of `h` returning an unexpected number of values can be ignored, because in that case the effect of the `let` is explicitly unspecified and implementation-dependent.)

APPENDICES

Appendix A. Formal semantics

This appendix presents a non-normative, formal, operational semantics for Scheme, that is based on an earlier semantics [18]. It does not cover the entire language. The notable missing features are the macro system, I/O, and the numerical tower. The precise list of features included is given in section A.2.

The core of the specification is a single-step term rewriting relation that indicates how an (abstract) machine behaves. In general, the report is not a complete specification, giving implementations freedom to behave differently, typically to allow optimizations. This underspecification shows up in two ways in the semantics.

The first is reduction rules that reduce to special "**unknown:** *string*" states (where the string provides a description of the unknown state). The intention is that rules that reduce to such states can be replaced with arbitrary reduction rules. The precise specification of how to replace those rules is given in section A.12.

The other is that the single-step relation relates one program to multiple different programs, each corresponding to a legal transition that an abstract machine might take. Accordingly we use the transitive closure of the single step relation $\rightarrow^*$ to define the semantics, $\mathcal{S}$, as a function from programs ($\mathcal{P}$) to sets of observable results ($\mathcal{R}$):

$$\mathcal{S} : \mathcal{P} \longrightarrow 2^{\mathcal{R}}$$
$$\mathcal{S}(\mathcal{P}) = \{\mathscr{O}(\mathcal{A}) \mid \mathcal{P} \rightarrow^* \mathcal{A}\}$$

where the function $\mathscr{O}$ turns an answer ($\mathcal{A}$) from the semantics into an observable result. Roughly, $\mathscr{O}$ is the identity function on simple base values, and returns a special tag for more complex values, like procedure and pairs.

So, an implementation conforms to the semantics if, for every program $\mathcal{P}$, the implementation produces one of the results in $\mathcal{S}(\mathcal{P})$ or, if the implementation loops forever, then there is an infinite reduction sequence starting at $\mathcal{P}$, assuming that the reduction relation $\rightarrow$ has been adjusted to replace the **unknown:** states.

The precise definitions of $\mathcal{P}$, $\mathcal{A}$, $\mathcal{R}$, and $\mathscr{O}$ are also given in section A.2.

To help understand the semantics and how it behaves, we have implemented it in PLT Redex. The implementation is available at the report's website: `http://www.r6rs.org/`. All of the reduction rules and the metafunctions shown in the figures in this semantics were generated automatically from the source code.

A.1. Background

We assume the reader has a basic familiarity with context-sensitive reduction semantics. Readers unfamiliar with this system may wish to consult Felleisen and Flatt's monograph [10] or Wright and Felleisen [29] for a thorough introduction, including the relevant technical background, or an introduction to PLT Redex [19] for a somewhat lighter one.

As a rough guide, we define the operational semantics of a language via a relation on program terms, where the relation corresponds to a single step of an abstract machine. The relation is defined using evaluation contexts, namely terms with a distinguished place in them, called *holes*, where the next step of evaluation occurs. We say that a term e decomposes into an evaluation context E and another term e' if e is the same as E but with the hole replaced by e'. We write $E[e']$ to indicate the term obtained by replacing the hole in E with e'.

For example, assuming that we have defined a grammar containing non-terminals for evaluation contexts (E), expressions (e), variables (x), and values (v), we would write:

$$E_1[\texttt{((lambda (}x_1 \cdots\texttt{)}\ e_1\texttt{)}\ v_1\ \cdots\texttt{)}] \rightarrow$$
$$E_1[\{x_1 \cdots \mapsto v_1 \cdots\}e_1] \qquad (\#x_1 = \#v_1)$$

to define the β_v rewriting rule (as a part of the $\rightarrow$ single step relation). We use the names of the non-terminals (possibly with subscripts) in a rewriting rule to restrict the application of the rule, so it applies only when some term produced by that grammar appears in the corresponding position in the term. If the same non-terminal with an identical subscript appears multiple times, the rule only applies when the corresponding terms are structurally identical (nonterminals without subscripts are not constrained to match each other). Thus, the occurrence of E_1 on both the left-hand and right-hand side of the rule above means that the context of the application expression does not change when using this rule. The ellipses are a form of Kleene star, meaning that zero or more occurrences of terms matching the pattern proceeding the ellipsis may appear in place of the the ellipsis and the pattern preceding it. We use the notation $\{x_1 \cdots \mapsto v_1 \cdots\}e_1$ for capture-avoiding substitution; in this case it means that each x_1 is replaced with the corresponding v_1 in e_1. Finally, we write side-conditions in parentheses beside a rule; the side-condition in the above rule indicates that the number of x_1s must be the same as the number of v_1s. Sometimes we use equality in the side-conditions; when we do it merely means simple term equality, i.e., the two terms must have the same syntactic shape.

Making the evaluation context E explicit in the rule allows us to define relations that manipulate their context. As a simple example, we can add another rule that signals a violation when a procedure is applied to the wrong number of arguments by discarding the evaluation context on the

```
P        ::= (store (sf ···) es)
          |  uncaught exception: v  |  unknown: description
A        ::= (store (sf ···) (values v ···))
          |  uncaught exception: v  |  unknown: description
R        ::= (values R_v ···)  |  exception  |  unknown
R_v      ::= pair  |  null  |  'sym  |  sqv  |  condition  |  procedure
sf       ::= (x v)  |  (x bh)  |  (pp (cons v v))
es       ::= 'seq  |  'sqv  |  '()  |  (begin es es ···)
          |  (begin0 es es ···)  |  (es es ···)  |  (if es es es)  |  (set! x es)  |  x  |  nonproc
          |  pproc  |  (lambda f es es ···)
          |  (letrec ((x es) ···) es es ···)
          |  (letrec* ((x es) ···) es es ···)
          |  (dw x es es es)  |  (throw x es)  |  unspecified  |  (handlers es ··· es)
          |  (l! x es)  |  (reinit x)
f        ::= (x ···)
          |  (x x ··· dot x)
          |  x
s        ::= seq  |  ()  |  sqv  |  sym
seq      ::= (s s ···)  |  (s s ··· dot sqv)  |  (s s ··· dot sym)
sqv      ::= n  |  #t  |  #f

p        ::= (store (sf ···) e)
e        ::= (begin e e ···)  |  (begin0 e e ···)  |  (e e ···)  |  (if e e e)  |  (set! x e)
          |  (handlers e ··· e)  |  x  |  nonproc  |  proc  |  (dw x e e e)  |  unspecified
          |  (letrec ((x e) ···) e e ···)
          |  (letrec* ((x e) ···) e e ···)
          |  (l! x es)  |  (reinit x)

v        ::= nonproc  |  proc
nonproc  ::= pp  |  null  |  'sym  |  sqv  |  (make-cond string)
proc     ::= (lambda f e e ···)  |  pproc  |  (throw x e)
pproc    ::= aproc  |  proc1  |  proc2  |  list  |  dynamic-wind  |  apply  |  values
proc1    ::= null?  |  pair?  |  car  |  cdr  |  call/cc  |  procedure?  |  condition?  |  raise*
proc2    ::= cons  |  consi  |  set-car!  |  set-cdr!  |  eqv?  |  call-with-values  |  with-exception-handler
aproc    ::= +  |  -  |  /  |  *
raise*   ::= raise-continuable  |  raise

pp       ::= ip  |  mp
ip       ::= [immutable pair pointers]
mp       ::= [mutable pair pointers]

sym      ::= [variables except dot]
x        ::= [variables except dot and keywords]
n        ::= [numbers]
```

Figure A.2a: Grammar for programs and observables

right-hand side of a rule:

$$E[\texttt{((lambda (}x_1 \cdots\texttt{) }e\texttt{) }v_1 \ \cdots\texttt{)}] \rightarrow$$
$$\textbf{violation:}\ \text{wrong argument count} \qquad (\#x_1 \neq \#v_1)$$

Later we take advantage of the explicit evaluation context in more sophisticated ways.

A.2. Grammar

Figure A.2a shows the grammar for the subset of the report this semantics models. Non-terminals are written in *italics* or in a calligraphic font ($\mathcal{P}$ $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{R}_v$) and literals are written in a `monospaced` font.

The $\mathcal{P}$ non-terminal represents possible program states. The first alternative is a program with a store and an ex-

```
P   ::= (store (sf ···) E*)

E   ::= F[(handlers proc ··· E*)]  |  F[(dw x e E* e)]  |  F
E*  ::= [ ]*  |  E
E°  ::= [ ]°  |  E

F   ::= [ ]  |  (v ··· F° v ···)  |  (if F° e e)  |  (set! x F°)  |  (begin F* e e ···)
     |  (begin0 F* e e ···)  |  (begin0 (values v ···) F* e ···)
     |  (begin0 unspecified F* e ···)  |  (call-with-values (lambda () F* e ···) v)
     |  (l! x F°)
F*  ::= [ ]*  |  F
F°  ::= [ ]°  |  F
U   ::= (v ··· [ ] v ···)  |  (if [ ] e e)  |  (set! x [ ])  |  (l! x [ ])
     |  (call-with-values (lambda () [ ]) v)

PG  ::= (store (sf ···) G)
G   ::= F[(dw x e G e)]  |  F
H   ::= F[(handlers proc ··· H)]  |  F

S   ::= [ ]  |  (begin e e ··· S es ···)  |  (begin S es ···)  |  (begin0 e e ··· S es ···)
     |  (begin0 S es ···)  |  (e ··· S es ···)  |  (if S es es)  |  (if e S es)  |  (if e e S)
     |  (set! x S)  |  (handlers s ··· S es ··· es)  |  (handlers s ··· S)  |  (throw x e)
     |  (lambda f S es ···)  |  (lambda f e e ··· S es ···)
     |  (letrec ((x e) ··· (x S) (x es) ···) es es ···)
     |  (letrec ((x e) ···) S es ···)  |  (letrec ((x e) ···) e e ··· S es ···)
     |  (letrec* ((x e) ··· (x S) (x es) ···) es es ···)
     |  (letrec* ((x e) ···) S es ···)  |  (letrec* ((x e) ···) e e ··· S es ···)
```

Figure A.2b: Grammar for evaluation contexts

pression. The second alternative is an uncaught exception, and the third is used to indicate a place where the model does not completely specify the behavior of the primitives it models (see section A.12 for details of those situations). The $\mathcal{A}$ non-terminal represents a final result of a program. It is just like $\mathcal{P}$ except that expression has been reduced to some sequence of values.

The $\mathcal{R}$ and $\mathcal{R}_v$ non-terminals specify the observable results of a program. Each $\mathcal{R}$ is either a sequence of values that correspond to the values produced by the program that terminates normally, or a tag indicating an uncaught exception was raised, or `unknown` if the program encounters a situation the semantics does not cover. The $\mathcal{R}_v$ non-terminal specifies what the observable results are for a particular value: a pair, the empty list, a symbol, a self-quoting value (`#t`, `#f`, and numbers), a condition, or a procedure.

The *sf* non-terminal generates individual elements of the store. The store holds all of the mutable state of a program. It is explained in more detail along with the rules that manipulate it.

Expressions (*es*) include quoted data, `begin` expressions, `begin0` expressions[1], application expressions, `if` expressions, `set!` expressions, variables, non-procedure values (*nonproc*), primitive procedures (*pproc*), lambda expressions, `letrec` and `letrec*` expressions.

The last few expression forms are only generated for intermediate states (`dw` for `dynamic-wind`, `throw` for continuations, `unspecified` for the result of the assignment operators, `handlers` for exception handlers, and `l!` and `reinit` for `letrec`), and should not appear in an initial program. Their use is described in the relevant sections of this appendix.

The *f* non-terminal describes the formals for `lambda` expressions. (The `dot` is used instead of a period for procedures that accept an arbitrary number of arguments, in

[1] `begin0` is not part of the standard, but we include it to make the rules for `dynamic-wind` and `letrec` easier to read. Although we model it directly, it can be defined in terms of other forms we model here that do come from the standard:

```
(begin0 e1 e2 ···)  =  (call-with-values
                         (lambda () e1)
                         (lambda x
                           e2 ···
                           (apply values x)))
```

order to avoid meta-circular confusion in our PLT Redex model.)

The s non-terminal covers all datums, which can be either non-empty sequences (seq), the empty sequence, self-quoting values (sqv), or symbols. Non-empty sequences are either just a sequence of datums, or they are terminated with a dot followed by either a symbol or a self-quoting value. Finally the self-quoting values are numbers and the booleans `#t` and `#f`.

The p non-terminal represents programs that have no quoted data. Most of the reduction rules rewrite p to p, rather than $\mathcal{P}$ to $\mathcal{P}$, since quoted data is first rewritten into calls to the list construction functions before ordinary evaluation proceeds. In parallel to es, e represents expressions that have no quoted expressions.

The values (v) are divided into four categories:

- Non-procedures ($nonproc$) include pair pointers (`pp`), the empty list (`null`), symbols, self-quoting values (sqv), and conditions. Conditions represent the report's condition values, but here just contain a message and are otherwise inert.
- User procedures (`(lambda` f e e $\cdots$`)`) include multi-arity lambda expressions and lambda expressions with dotted parameter lists,
- Primitive procedures ($pproc$) include
  - arithmetic procedures ($aproc$): `+`, `-`, `/`, and `*`,
  - procedures of one argument ($proc1$): `null?`, `pair?`, `car`, `cdr`, `call/cc`, `procedure?`, `condition?`, `unspecified?`, `raise`, and `raise-continuable`,
  - procedures of two arguments ($proc2$): `cons`, `set-car!`, `set-cdr!`, `eqv?`, and `call-with-values`,
  - as well as `list`, `dynamic-wind`, `apply`, `values`, and `with-exception-handler`.
- Finally, continuations are represented as `throw` expressions whose body consists of the context where the continuation was grabbed.

The next three set of non-terminals in figure A.2a represent pairs (pp), which are divided into immutable pairs (ip) and mutable pairs (mp). The final set of non-terminals in figure A.2a, sym, x, and n represent symbols, variables, and numbers respectively. The non-terminals ip, mp, and sym are all assumed to all be disjoint. Additionally, the variables x are assumed not to include any keywords or primitive operations, so any program variables whose names coincide with them must be renamed before the semantics can give the meaning of that program.

The set of non-terminals for evaluation contexts is shown in figure A.2b. The P non-terminal controls where evaluation happens in a program that does not contain any quoted data. The E and F evaluation contexts are for expressions. They are factored in that manner so that the PG, G, and H evaluation contexts can re-use F and have fine-grained control over the context to support exceptions and `dynamic-wind`. The starred and circled variants, $E^\star$, E°, $F^\star$, and F° dictate where a single value is promoted to multiple values and where multiple values are demoted to a single value. The U context is used to manage the report's underspecification of the results of `set!`, `set-car!`, and `set-cdr!` (see section A.12 for details). Finally, the S context is where quoted expressions can be simplified. The precise use of the evaluation contexts is explained along with the relevant rules.

Although it is not written in the grammar figure, variable sequences bound in the store, and in `lambda`, `letrec`, and `letrec*` must not contain any duplicates.

To convert the answers ($\mathcal{A}$) of the semantics into observable results, we use these two functions:

$$\begin{array}{l}
\mathscr{O} : \mathcal{A} \rightarrow \mathcal{R} \\
\mathscr{O}[\![\texttt{(store (}sf\ \cdots\texttt{) (values } v_1\ \cdots\texttt{))}]\!] = \\
\quad \texttt{(values } \mathscr{O}_v[\![v_1]\!]\ \cdots\texttt{)} \\[1ex]
\mathscr{O}[\![\textbf{uncaught exception: } v]\!] = \\
\quad \texttt{exception} \\[1ex]
\mathscr{O}[\![\textbf{unknown: } \textit{description}]\!] = \\
\quad \texttt{unknown}
\end{array}$$

$$\begin{array}{lcl}
\mathscr{O}_v : v \rightarrow \mathcal{R}_v & & \\
\mathscr{O}_v[\![pp_1]\!] & = & \texttt{pair} \\
\mathscr{O}_v[\![\texttt{null}]\!] & = & \texttt{null} \\
\mathscr{O}_v[\![{'sym_1}]\!] & = & {'sym_1} \\
\mathscr{O}_v[\![sqv_1]\!] & = & sqv_1 \\
\mathscr{O}_v[\![\texttt{(make-cond } string\texttt{)}]\!] & = & \texttt{condition} \\
\mathscr{O}_v[\![proc]\!] & = & \texttt{procedure}
\end{array}$$

They eliminate the store, and replace complex values with simple tags that indicate only the kind of value that was produced or, if no values were produced, indicates that either an uncaught exception was raised, or that the program reached a state that is not specified by the semantics.

A.3. Quote

The first reduction rules that apply to any program is the rules in figure A.3 that eliminate quoted expressions. The first two rules erase the quote for quoted expressions that do not introduce any pairs. The last two rules lift quoted datums to the top of the expression so they are evaluated only once, and turn the datums into calls to either `cons` or `consi`, via the metafunctions $\mathscr{Q}_i$ and $\mathscr{Q}_m$.

$$\texttt{(store (}sf_1 \cdots\texttt{)}\ S_1[{}'sqv_1]\texttt{)} \rightarrow \qquad \text{[6sqv]}$$
$$\texttt{(store (}sf_1 \cdots\texttt{)}\ S_1[sqv_1]\texttt{)}$$

$$\texttt{(store (}sf_1 \cdots\texttt{)}\ S_1[{}'\texttt{()}]\texttt{)} \rightarrow \qquad \text{[6eseq]}$$
$$\texttt{(store (}sf_1 \cdots\texttt{)}\ S_1[\texttt{null}]\texttt{)}$$

$$\texttt{(store (}sf_1 \cdots\texttt{)}\ S_1[{}'seq_1]\texttt{)} \rightarrow \qquad \text{[6qcons]}$$
$$\texttt{(store (}sf_1 \cdots\texttt{) ((lambda (}qp\texttt{)}\ S_1[qp]\texttt{)}\ \mathscr{Q}_i[\![seq_1]\!]\texttt{))} \quad (qp \text{ fresh})$$

$$\texttt{(store (}sf_1 \cdots\texttt{)}\ S_1[{}'seq_1]\texttt{)} \rightarrow \qquad \text{[6qconsi]}$$
$$\texttt{(store (}sf_1 \cdots\texttt{) ((lambda (}qp\texttt{)}\ S_1[qp]\texttt{)}\ \mathscr{Q}_m[\![seq_1]\!]\texttt{))} \quad (qp \text{ fresh})$$

$$
\begin{array}{lcl}
\mathscr{Q}_i : seq \rightarrow e & & \\
\mathscr{Q}_i[\![\texttt{()}]\!] & = & \texttt{null} \\
\mathscr{Q}_i[\![\texttt{(}s_1\ s_2\ \cdots\texttt{)}]\!] & = & \texttt{(cons}\ \mathscr{Q}_i[\![s_1]\!]\ \mathscr{Q}_i[\![\texttt{(}s_2\ \cdots\texttt{)}]\!]\texttt{)} \\
\mathscr{Q}_i[\![\texttt{(}s_1\ \texttt{dot}\ sqv_1\texttt{)}]\!] & = & \texttt{(cons}\ \mathscr{Q}_i[\![s_1]\!]\ sqv_1\texttt{)} \\
\mathscr{Q}_i[\![\texttt{(}s_1\ s_2\ s_3\ \cdots\ \texttt{dot}\ sqv_1\texttt{)}]\!] & = & \texttt{(cons}\ \mathscr{Q}_i[\![s_1]\!]\ \mathscr{Q}_i[\![\texttt{(}s_2\ s_3\ \cdots\ \texttt{dot}\ sqv_1\texttt{)}]\!]\texttt{)} \\
\mathscr{Q}_i[\![\texttt{(}s_1\ \texttt{dot}\ sym_1\texttt{)}]\!] & = & \texttt{(cons}\ \mathscr{Q}_i[\![s_1]\!]\ {}'sym_1\texttt{)} \\
\mathscr{Q}_i[\![\texttt{(}s_1\ s_2\ s_3\ \cdots\ \texttt{dot}\ sym_1\texttt{)}]\!] & = & \texttt{(cons}\ \mathscr{Q}_i[\![s_1]\!]\ \mathscr{Q}_i[\![\texttt{(}s_2\ s_3\ \cdots\ \texttt{dot}\ sym_1\texttt{)}]\!]\texttt{)} \\
\mathscr{Q}_i[\![sym_1]\!] & = & {}'sym_1 \\
\mathscr{Q}_i[\![sqv_1]\!] & = & sqv_1
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{Q}_m : seq \rightarrow e & & \\
\mathscr{Q}_m[\![\texttt{()}]\!] & = & \texttt{null} \\
\mathscr{Q}_m[\![\texttt{(}s_1\ s_2\ \cdots\texttt{)}]\!] & = & \texttt{(consi}\ \mathscr{Q}_m[\![s_1]\!]\ \mathscr{Q}_m[\![\texttt{(}s_2\ \cdots\texttt{)}]\!]\texttt{)} \\
\mathscr{Q}_m[\![\texttt{(}s_1\ \texttt{dot}\ sqv_1\texttt{)}]\!] & = & \texttt{(consi}\ \mathscr{Q}_m[\![s_1]\!]\ sqv_1\texttt{)} \\
\mathscr{Q}_m[\![\texttt{(}s_1\ s_2\ s_3\ \cdots\ \texttt{dot}\ sqv_1\texttt{)}]\!] & = & \texttt{(consi}\ \mathscr{Q}_m[\![s_1]\!]\ \mathscr{Q}_m[\![\texttt{(}s_2\ s_3\ \cdots\ \texttt{dot}\ sqv_1\texttt{)}]\!]\texttt{)} \\
\mathscr{Q}_m[\![\texttt{(}s_1\ \texttt{dot}\ sym_1\texttt{)}]\!] & = & \texttt{(consi}\ \mathscr{Q}_m[\![s_1]\!]\ {}'sym_1\texttt{)} \\
\mathscr{Q}_m[\![\texttt{(}s_1\ s_2\ s_3\ \cdots\ \texttt{dot}\ sym_1\texttt{)}]\!] & = & \texttt{(consi}\ \mathscr{Q}_m[\![s_1]\!]\ \mathscr{Q}_m[\![\texttt{(}s_2\ s_3\ \cdots\ \texttt{dot}\ sym_1\texttt{)}]\!]\texttt{)} \\
\mathscr{Q}_m[\![sym_1]\!] & = & {}'sym_1 \\
\mathscr{Q}_m[\![sqv_1]\!] & = & sqv_1
\end{array}
$$

Figure A.3: Quote

$$P_1[v_1]_\star \rightarrow \qquad \text{[6promote]}$$
$$P_1[\texttt{(values}\ v_1\texttt{)}]$$

$$P_1[\texttt{(values}\ v_1\texttt{)}]_\circ \rightarrow \qquad \text{[6demote]}$$
$$P_1[v_1]$$

$$P_1[\texttt{(call-with-values (lambda () (values}\ v_2\ \cdots\texttt{))}\ v_1\texttt{)}] \rightarrow \qquad \text{[6cwvd]}$$
$$P_1[\texttt{(}v_1\ v_2\ \cdots\texttt{)}]$$

$$P_1[\texttt{(call-with-values}\ v_1\ v_2\texttt{)}] \rightarrow \qquad \text{[6cwvw]}$$
$$P_1[\texttt{(call-with-values (lambda () (}v_1\texttt{))}\ v_2\texttt{)}] \quad (v_1 \neq \texttt{(lambda ()}\ e\texttt{)})$$

Figure A.4: Multiple values and call-with-values

Note that the left-hand side of the [6qcons] and [6qconsi] rules are identical, meaning that if one rule applies to a term, so does the other rule. Accordingly, a quoted expression may be lifted out into a sequence of `cons` expressions, which create mutable pairs, or into a sequence of `consi` expressions, which create immutable pairs (see section A.7 for the rules on how that happens).

These rules apply before any other because of the contexts in which they, and all of the other rules, apply. In particular, these rule applies in the S context. Figure A.2b shows that the S context allows this reduction to apply in any subexpression of an e, as long as all of the subexpressions to the left have no quoted expressions in them, although expressions to the right may have quoted expressions. Accordingly, this rule applies once for each quoted expression in the program, moving out to the beginning of the program. The rest of the rules apply in contexts that do not contain any quoted expressions, ensuring that these rules convert all quoted data into lists before those rules apply.

Although the identifier *qp* does not have a subscript, the semantics of PLT Redex's "fresh" declaration takes special care to ensures that the *qp* on the right-hand side of the rule is indeed the same as the one in the side-condition.

A.4. Multiple values

The basic strategy for multiple values is to add a rule that demotes (`values` v) to v and another rule that promotes v to (`values` v). If we allowed these rules to apply in an arbitrary evaluation context, however, we would get infinite reduction sequences of endless alternation between promotion and demotion. So, the semantics allows demotion only in a context expecting a single value and allows promotion only in a context expecting multiple values. We obtain this behavior with a small extension to the Felleisen-Hieb framework (also present in the operational model for R^{5}RS [17]). We extend the notation so that holes have names (written with a subscript), and the context-matching syntax may also demand a hole of a particular name (also written with a subscript, for instance $E[e]_\star$). The extension allows us to give different names to the holes in which multiple values are expected and those in which single values are expected, and structure the grammar of contexts accordingly.

To exploit this extension, we use three kinds of holes in the evaluation context grammar in figure A.2b. The ordinary hole $[\ ]$ appears where the usual kinds of evaluation can occur. The hole $[\ ]_\star$ appears in contexts that allow multiple values and $[\ ]_\circ$ appears in contexts that expect a single value. Accordingly, the rule [6promote] only applies in $[\ ]_\star$ contexts, and [6demote] only applies in $[\ ]_\circ$ contexts.

To see how the evaluation contexts are organized to ensure that promotion and demotion occur in the right places, consider the F, $F^\star$ and F° evaluation contexts. The $F^\star$ and F° evaluation contexts are just the same as F, except that they allow promotion to multiple values and demotion to a single value, respectively. So, the F evaluation context, rather than being defined in terms of itself, exploits $F^\star$ and F° to dictate where promotion and demotion can occur. For example, F can be (`if` F° e e) meaning that demotion from (`values` v) to v can occur in the test of an `if` expression. Similarly, F can be (`begin` $F^\star$ e e $\cdots$) meaning that v can be promoted to (`values` v) in the first subexpression of a `begin`.

In general, the promotion and demotion rules simplify the definitions of the other rules. For instance, the rule for `if` does not need to consider multiple values in its first subexpression. Similarly, the rule for `begin` does not need to consider the case of a single value as its first subexpression.

The other two rules in figure A.4 handle `call-with-values`. The evaluation contexts for `call-with-values` (in the F non-terminal) allow evaluation in the body of a procedure that has been passed as the first argument to `call-with-values`, as long as the second argument has been reduced to a value. Once evaluation inside that procedure completes, it will produce multiple values (since it is an $F^\star$ position), and the entire `call-with-values` expression reduces to an application of its second argument to those values, via the rule [6cwvd]. Finally, in the case that the first argument to `call-with-values` is a value, but is not of the form (`lambda () ` e), the rule [6cwvw] wraps it in a thunk to trigger evaluation.

A.5. Exceptions

The workhorses for the exception system are

(`handlers` *proc* $\cdots$ e)

expressions and the G and PG evaluation contexts (shown in figure A.2b). The `handlers` expression records the active exception handlers (*proc* $\cdots$) in some expression (e). The intention is that only the nearest enclosing `handlers` expression is relevant to raised exceptions, and the G and PG evaluation contexts help achieve that goal. They are just like their counterparts E and P, except that `handlers` expressions cannot occur on the path to the hole, and the exception system rules take advantage of that context to find the closest enclosing handler.

To see how the contexts work together with `handler` expressions, consider the left-hand side of the [6xunee] rule in figure A.5. It matches expressions that have a call to `raise` or `raise-continuable` (the non-terminal *raise** matches both exception-raising procedures) in a PG evaluation context. Since the PG context does not contain any `handlers` expressions, this exception cannot be caught, so this expression reduces to a final state indicating the uncaught exception. The rule [6xuneh] also signals an uncaught exception, but it covers the case where a `handlers` expression has exhausted all of the handlers available to it. The rule applies to expressions that have a `handlers` expression (with no exception handlers) in an arbitrary evaluation context where a call to one of the exception-raising functions is nested in the `handlers` expression. The use of the G evaluation context ensures that there are no other `handler` expressions between this one and the raise.

The next two rules cover call to the procedure `with-exception-handler`. The [6xwh1] rule applies when there are no `handler` expressions. It constructs a new one

$PG[(raise{*}\ v_1)] \rightarrow$ [6xunee]
uncaught exception: v_1

$P[$(handlers $G[(raise{*}\ v_1)]$)$] \rightarrow$ [6xuneh]
uncaught exception: v_1

$PG_1[$(with-exception-handler $proc_1$ $proc_2$)$] \rightarrow$ [6xwh1]
$PG_1[$(handlers $proc_1$ ($proc_2$))$]$

$P_1[$(handlers $proc_1 \cdots G_1[$(with-exception-handler $proc_2$ $proc_3$)$]$)$] \rightarrow$ [6xwhn]
$P_1[$(handlers $proc_1 \cdots G_1[$(handlers $proc_1 \cdots proc_2$ ($proc_3$))$]$)$]$

$P_1[$(handlers $proc_1 \cdots G_1[$(with-exception-handler v_1 v_2)$]$)$] \rightarrow$ [6xwhne]
$P_1[$(handlers $proc_1 \cdots G_1[$(raise (make-cond "with-exception-handler expects procs"))$]$)$]$ ($v_1 \notin proc$ or $v_2 \notin proc$)

$P_1[$(handlers $proc_1 \cdots proc_2$ $G_1[$(raise-continuable v_1)$]$)$] \rightarrow$ [6xrc]
$P_1[$(handlers $proc_1 \cdots proc_2$ $G_1[$(handlers $proc_1 \cdots$ ($proc_2$ v_1))$]$)$]$

$P_1[$(handlers $proc_1 \cdots proc_2$ $G_1[$(raise v_1)$]$)$] \rightarrow$ [6xr]
$P_1[$(handlers $proc_1 \cdots proc_2$
$G_1[$(handlers $proc_1 \cdots$ (begin ($proc_2$ v_1) (raise (make-cond "handler returned"))))$]$)$]$

$P_1[$(condition? (make-cond $string$))$] \rightarrow$ [6ct]
$P_1[$#t$]$

$P_1[$(condition? v_1)$] \rightarrow$ [6cf]
$P_1[$#f$]$ ($v_1 \neq$ (make-cond $string$))

$P_1[$(handlers $proc_1 \cdots$ (values $v_1 \cdots$))$] \rightarrow$ [6xdone]
$P_1[$(values $v_1 \cdots$)$]$

$PG_1[$(with-exception-handler v_1 v_2)$] \rightarrow$ [6weherr]
$PG_1[$(raise (make-cond "with-exception-handler expects procs"))$]$ ($v_1 \notin proc$ or $v_2 \notin proc$)

Figure A.5: Exceptions

and applies v_2 as a thunk in the `handler` body. If there already is a handler expression, the [6xwhn] applies. It collects the current handlers and adds the new one into a new `handlers` expression and, as with the previous rule, invokes the second argument to `with-exception-handlers`.

The next two rules cover exceptions that are raised in the context of a `handlers` expression. If a continuable exception is raised, [6xrc] applies. It takes the most recently installed handler from the nearest enclosing `handlers` expression and applies it to the argument to `raise-continuable`, but in a context where the exception handlers do not include that latest handler. The [6xr] rule behaves similarly, except it raises a new exception if the handler returns. The new exception is created with the `make-cond` special form.

The `make-cond` special form is a stand-in for the report's conditions. It does not evaluate its argument (note its absence from the E grammar in figure A.2b). That argument is just a literal string describing the context in which the exception was raised. The only operation on conditions is `condition?`, whose semantics are given by the two rules [6ct] and [6cf].

Finally, the rule [6xdone] drops a `handlers` expression when its body is fully evaluated, and the rule [6weherr] raises an exception when `with-exception-handler` is supplied with incorrect arguments.

A.6. Arithmetic and basic forms

This model does not include the report's arithmetic, but does include an idealized form in order to make experimentation with other features and writing test suites for the model simpler. Figure A.6 shows the reduction rules for the primitive procedures that implement addition, subtraction, multiplication, and division. They defer to their mathematical analogues. In addition, when the subtraction or divison operator are applied to no arguments, or when division receives a zero as a divisor, or when any of the arithmetic operations receive a non-number, an exception is raised.

The bottom half of figure A.6 shows the rules for `if`, `begin`, and `begin0`. The relevant evaluation contexts are given by the F non-terminal.

$P_1[\texttt{(+)}]$	$\rightarrow P_1[0]$	[6+0]
$P_1[\texttt{(+ } n_1\ n_2\ \cdots\texttt{)}]$	$\rightarrow P_1[\ulcorner\Sigma\{n_1, n_2\cdots\}\urcorner]$	[6+]
$P_1[\texttt{(- } n_1\texttt{)}]$	$\rightarrow P_1[\ulcorner -n_1\urcorner]$	[6u-]
$P_1[\texttt{(- } n_1\ n_2\ n_3\ \cdots\texttt{)}]$	$\rightarrow P_1[\ulcorner n_1 - \Sigma\{n_2, n_3\cdots\}\urcorner]$	[6-]
$P_1[\texttt{(-)}]$	$\rightarrow P_1[\texttt{(raise (make-cond }$ "arity mismatch"$\texttt{))}]$	[6-arity]
$P_1[\texttt{(*)}]$	$\rightarrow P_1[1]$	[6*1]
$P_1[\texttt{(* } n_1\ n_2\ \cdots\texttt{)}]$	$\rightarrow P_1[\ulcorner\Pi\{n_1, n_2\cdots\}\urcorner]$	[6*]
$P_1[\texttt{(/ } n_1\texttt{)}]$	$\rightarrow P_1[\texttt{(/ } 1\ n_1\texttt{)}]$	[6u/]
$P_1[\texttt{(/ } n_1\ n_2\ n_3\ \cdots\texttt{)}]$	$\rightarrow P_1[\ulcorner n_1/\Pi\{n_2, n_3\cdots\}\urcorner]$ $(0 \notin \{n_2, n_3, \ldots\})$	[6/]
$P_1[\texttt{(/ } n_1\ n_2\ \cdots\ 0\ n_3\ \cdots\texttt{)}]$	$\rightarrow P_1[\texttt{(raise (make-cond }$ "divison by zero"$\texttt{))}]$	[6/0]
$P_1[\texttt{(/)}]$	$\rightarrow P_1[\texttt{(raise (make-cond }$ "arity mismatch"$\texttt{))}]$	[6/arity]
$P_1[\texttt{(}aproc\ v_1\ \cdots\texttt{)}]$	$\rightarrow P_1[\texttt{(raise (make-cond }$ "arith-op applied to non-number"$\texttt{))}]$ $(\exists v \in v_1 \cdots$ s.t. v is not a number$)$	[6ae]

$P_1[\texttt{(if } v_1\ e_1\ e_2\texttt{)}]$	$\rightarrow P_1[e_1]$ $(v_1 \neq \texttt{\#f})$	[6if3t]
$P_1[\texttt{(if \#f } e_1\ e_2\texttt{)}]$	$\rightarrow P_1[e_2]$	[6if3f]
$P_1[\texttt{(begin (values } v\ \cdots\texttt{) } e_1\ e_2\ \cdots\texttt{)}]$	$\rightarrow P_1[\texttt{(begin } e_1\ e_2\ \cdots\texttt{)}]$	[6beginc]
$P_1[\texttt{(begin } e_1\texttt{)}]$	$\rightarrow P_1[e_1]$	[6begind]
$P_1[\texttt{(begin0 (values } v_1\ \cdots\texttt{) (values } v_2\ \cdots\texttt{) } e_2\ \cdots\texttt{)}]$	$\rightarrow P_1[\texttt{(begin0 (values } v_1\ \cdots\texttt{) } e_2\ \cdots\texttt{)}]$	[6begin0n]
$P_1[\texttt{(begin0 } e_1\texttt{)}]$	$\rightarrow P_1[e_1]$	[6begin01]

Figure A.6: Arithmetic and basic forms

The evaluation contexts for `if` only allow evaluation in its test expression. Once that is a value, the rules reduce an `if` expression to its consequent if the test is not `#f`, and to its alternative if it is `#f`.

The `begin` evaluation contexts allow evaluation in the first subexpression of a begin, but only if there are two or more subexpressions. In that case, once the first expression has been fully simplified, the reduction rules drop its value. If there is only a single subexpression, the `begin` itself is dropped.

Like the `begin` evaluation contexts, the `begin0` evaluation contexts allow evaluation of the first subexpression of a `begin0` expression when there are two or more subexpressions. The `begin0` evaluation contexts also allow evaluation in the second subexpression of a `begin0` expression, as long as the first subexpression has been fully simplified. The [6begin0n] rule for `begin0` then drops a fully simplified second subexpression. Eventually, there is only a single expression in the `begin0`, at which point the [begin01] rule fires, and removes the `begin0` expression.

A.7. Lists

The rules in figure A.7 handle lists. The first two rules handle `list` by reducing it to a succession of calls to `cons`, followed by `null`.

The next two rules, [6cons] and [6consi], allocate new `cons` cells. They both move `(cons` v_1 v_2`)` into the store, bound to a fresh pair pointer (see also section A.3 for a description of "fresh"). The [6cons] uses a *mp* variable, to indicate the pair is mutable, and the [6consi] uses a *ip* variable to indicate the pair is immutable.

The rules [6car] and [6cdr] extract the components of a pair from the store when presented with a pair pointer (the *pp* can be either *mp* or *ip*, as shown in figure A.2a).

The rules [6setcar] and [6setcdr] handle assignment of mutable pairs. They replace the contents of the appropriate location in the store with the new value, and reduce to `unspecified`. See section A.12 for an explanation of how `unspecified` reduces.

P_1[(list v_1 v_2 $\cdots$)]→ [6listc]
P_1[(cons v_1 (list v_2 $\cdots$))]

P_1[(list)]→ [6listn]
P_1[null]

(store (sf_1 $\cdots$) E_1[(cons v_1 v_2)])→ [6cons]
(store (sf_1 $\cdots$ (mp (cons v_1 v_2))) E_1[mp]) (mp fresh)

(store (sf_1 $\cdots$) E_1[(consi v_1 v_2)])→ [6consi]
(store (sf_1 $\cdots$ (ip (cons v_1 v_2))) E_1[ip]) (ip fresh)

(store (sf_1 $\cdots$ (pp_i (cons v_1 v_2)) sf_2 $\cdots$) E_1[(car pp_i)])→ [6car]
(store (sf_1 $\cdots$ (pp_i (cons v_1 v_2)) sf_2 $\cdots$) E_1[v_1])

(store (sf_1 $\cdots$ (pp_i (cons v_1 v_2)) sf_2 $\cdots$) E_1[(cdr pp_i)])→ [6cdr]
(store (sf_1 $\cdots$ (pp_i (cons v_1 v_2)) sf_2 $\cdots$) E_1[v_2])

(store (sf_1 $\cdots$ (mp_1 (cons v_1 v_2)) sf_2 $\cdots$) E_1[(set-car! mp_1 v_3)])→ [6setcar]
(store (sf_1 $\cdots$ (mp_1 (cons v_3 v_2)) sf_2 $\cdots$) E_1[unspecified])

(store (sf_1 $\cdots$ (mp_1 (cons v_1 v_2)) sf_2 $\cdots$) E_1[(set-cdr! mp_1 v_3)])→ [6setcdr]
(store (sf_1 $\cdots$ (mp_1 (cons v_1 v_3)) sf_2 $\cdots$) E_1[unspecified])

P_1[(null? null)]→ [6null?t]
P_1[#t]

P_1[(null? v_1)]→ [6null?f]
P_1[#f] ($v_1 \neq$ null)

P_1[(pair? pp)]→ [6pair?t]
P_1[#t]

P_1[(pair? v_1)]→ [6pair?f]
P_1[#f] ($v_1 \notin pp$)

P_1[(car v_i)]→ [6care]
P_1[(raise (make-cond "can't take car of non-pair"))] ($v_i \notin pp$)

P_1[(cdr v_i)]→ [6cdre]
P_1[(raise (make-cond "can't take cdr of non-pair"))] ($v_i \notin pp$)

P_1[(set-car! v_1 v_2)]→ [6scare]
P_1[(raise (make-cond "can't set-car! on a non-pair or an immutable pair"))] ($v_1 \notin mp$)

P_1[(set-cdr! v_1 v_2)]→ [6scdre]
P_1[(raise (make-cond "can't set-cdr! on a non-pair or an immutable pair"))] ($v_1 \notin mp$)

Figure A.7: Lists

The next four rules handle the null? predicate and the pair? predicate, and the final four rules raise exceptions when car, cdr, set-car! or set-cdr! receive non pairs.

A.8. Eqv

The rules for eqv? are shown in figure A.8. The first two rules cover most of the behavior of eqv?. The first says that when the two arguments to eqv? are syntactically identical, then eqv? produces #t and the second says that when the arguments are not syntactically identical, then eqv? produces #f. The structure of v has been carefully designed so that simple term equality corresponds closely to eqv?'s behavior. For example, pairs are represented as pointers into the store and eqv? only compares those pointers.

The side-conditions on those first two rules ensure that they do not apply when simple term equality does not match the behavior of eqv?. There are two situations where it does not match: comparing two conditions and comparing two procedures. For the first, the report does not specify eqv?'s behavior, except to say that it must return a boolean, so the remaining two rules ([6eqct], and [6eqcf]) allow such comparisons to return #t or #f. Com-

P_1[`(eqv?` v_1 v_1`)`]→ [6eqt]
P_1[`#t`] ($v_1 \notin \mathit{proc}, v_1 \neq$ (`make-cond` *string*))

P_1[`(eqv?` v_1 v_2`)`]→ [6eqf]
P_1[`#f`] ($v_1 \neq v_2, v_1 \notin \mathit{proc}$ or $v_2 \notin \mathit{proc}, v_1 \neq$ (`make-cond` *string*) or $v_2 \neq$ (`make-cond` *string*))

P_1[`(eqv? (make-cond` string_1`) (make-cond` string_2`))`]→ [6eqct]
P_1[`#t`]

P_1[`(eqv? (make-cond` string_1`) (make-cond` string_2`))`]→ [6eqcf]
P_1[`#f`]

Figure A.8: Eqv

P_1[`(`e_1 ⋯ e_i e_{i+1} ⋯`)`]→ [6mark]
P_1[`((lambda (`x`) (`e_1 ⋯ x e_{i+1} ⋯`))` e_i`)`] (x fresh, $e_i \notin v, \exists e \in e_1 \cdots e_{i+1} \cdots$ s.t. $e \notin v$)

`(store (`sf_1 ⋯`)` E_1[`((lambda (`x_1 x_2 ⋯`)` e_1 e_2 ⋯`)` v_1 v_2 ⋯`)`]`)`→ [6appN!]
`(store (`sf_1 ⋯ `(`bp v_1`))` E_1[`(`$\{x_1 \mapsto \mathit{bp}\}$`(lambda (`$x_2$ ⋯`)` e_1 e_2 ⋯`)` v_2 ⋯`)`]`)`
(bp fresh, $\#x_2 = \#v_2, \mathscr{V}[\![x_1,$ `(lambda (`x_2 ⋯`)` e_1 e_2 ⋯`)`$]\!]$)

P_1[`((lambda (`x_1 x_2 ⋯`)` e_1 e_2 ⋯`)` v_1 v_2 ⋯`)`]→ [6appN]
P_1[`(`$\{x_1 \mapsto v_1\}$`(lambda (`x_2 ⋯`)` e_1 e_2 ⋯`)` v_2 ⋯`)`] ($\#x_2 = \#v_2, \neg\mathscr{V}[\![x_1,$ `(lambda (`x_2 ⋯`)` e_1 e_2 ⋯`)`$]\!]$)

P_1[`((lambda ()` e_1 e_2 ⋯`))`]→ [6app0]
P_1[`(begin` e_1 e_2 ⋯`)`]

P_1[`((lambda (`x_1 x_2 ⋯ `dot` x_r`)` e_1 e_2 ⋯`)` v_1 v_2 ⋯ v_3 ⋯`)`]→ [6μapp]
P_1[`((lambda (`x_1 x_2 ⋯ x_r`)` e_1 e_2 ⋯`)` v_1 v_2 ⋯ `(list` v_3 ⋯`))`] ($\#x_2 = \#v_2$)

P_1[`((lambda` x_1 e_1 e_2 ⋯`)` v_1 ⋯`)`]→ [6μapp1]
P_1[`((lambda (`x_1`)` e_1 e_2 ⋯`) (list` v_1 ⋯`))`]

`(store (`sf_1 ⋯ `(`x_1 v_1`)` sf_2 ⋯`)` E_1[x_1]`)`→ [6var]
`(store (`sf_1 ⋯ `(`x_1 v_1`)` sf_2 ⋯`)` E_1[v_1]`)`

`(store (`sf_1 ⋯ `(`x_1 v_1`)` sf_2 ⋯`)` E_1[`(set!` x_1 v_2`)`]`)`→ [6set]
`(store (`sf_1 ⋯ `(`x_1 v_2`)` sf_2 ⋯`)` E_1[`unspecified`]`)`

P_1[`(procedure?` *proc*`)`]→ [6proct]
P_1[`#t`]

P_1[`(procedure?` *nonproc*`)`]→ [6procf]
P_1[`#f`]

P_1[`((lambda (`x_1 ⋯`)` e_1 e_2 ⋯`)` v_1 ⋯`)`]→ [6arity]
P_1[`(raise (make-cond` "arity mismatch"`))`] ($\#x_1 \neq \#v_1$)

P_1[`((lambda (`x_1 x_2 ⋯ `dot` x`)` e_1 e_2 ⋯`)` v_1 ⋯`)`]→ [6μarity]
P_1[`(raise (make-cond` "arity mismatch"`))`] ($\#v_1 < \#x_2 + 1$)

P_1[`(`*nonproc* v ⋯`)`]→ [6appe]
P_1[`(raise (make-cond` "can't call non-procedure"`))`]

P_1[`(`*proc1* v_1 ⋯`)`]→ [61arity]
P_1[`(raise (make-cond` "arity mismatch"`))`] ($\#v_1 \neq 1$)

P_1[`(`*proc2* v_1 ⋯`)`]→ [62arity]
P_1[`(raise (make-cond` "arity mismatch"`))`] ($\#v_1 \neq 2$)

Figure A.9a: Procedures & application

$\mathscr{V} \in 2^{x \times e}$

$\mathscr{V}[\![x_1, \texttt{(set!}\ x_2\ e_1\texttt{)}]\!]$	if $x_1 = x_2$
$\mathscr{V}[\![x_1, \texttt{(set!}\ x_2\ e_1\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, e_1]\!]$ and $x_1 \neq x_2$
$\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ e_2\ e_3\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, e_1]\!]$ or $\mathscr{V}[\![x_1, \texttt{(begin}\ e_2\ e_3\ \cdots\texttt{)}]\!]$
$\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, e_1]\!]$
$\mathscr{V}[\![x_1, \texttt{(}e_1\ e_2\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ e_2\ \cdots\texttt{)}]\!]$
$\mathscr{V}[\![x_1, \texttt{(if}\ e_1\ e_2\ e_3\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, e_1]\!]$ or $\mathscr{V}[\![x_1, e_2]\!]$ or $\mathscr{V}[\![x_1, e_3]\!]$
$\mathscr{V}[\![x_1, \texttt{(begin0}\ e_1\ e_2\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ e_2\ \cdots\texttt{)}]\!]$
$\mathscr{V}[\![x_1, \texttt{(lambda (}x_2\ \cdots\texttt{)}\ e_1\ e_2\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ e_2\ \cdots\texttt{)}]\!]$ and $x_1 \notin \{x_2 \cdots\}$
$\mathscr{V}[\![x_1, \texttt{(lambda (}x_2\ \cdots\ \texttt{dot}\ x_3\texttt{)}\ e_1\ e_2\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ e_2\ \cdots\texttt{)}]\!]$ and $x_1 \notin \{x_2 \cdots x_3\}$
$\mathscr{V}[\![x_1, \texttt{(lambda}\ x_2\ e_1\ e_2\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ e_2\ \cdots\texttt{)}]\!]$ and $x_1 \neq x_2$
$\mathscr{V}[\![x_1, \texttt{(letrec ((}x_2\ e_1\texttt{)}\ \cdots\texttt{)}\ e_2\ e_3\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ \cdots\ e_2\ e_3\ \cdots\texttt{)}]\!]$ and $x_1 \notin \{x_2 \cdots\}$
$\mathscr{V}[\![x_1, \texttt{(letrec* ((}x_2\ e_1\texttt{)}\ \cdots\texttt{)}\ e_2\ e_3\ \cdots\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(begin}\ e_1\ \cdots\ e_2\ e_3\ \cdots\texttt{)}]\!]$ and $x_1 \notin \{x_2 \cdots\}$
$\mathscr{V}[\![x_1, \texttt{(l!}\ x_2\ e_1\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(set!}\ x_2\ e_1\texttt{)}]\!]$
$\mathscr{V}[\![x_1, \texttt{(reinit}\ x_2\ e_1\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, \texttt{(set!}\ x_2\ e_1\texttt{)}]\!]$
$\mathscr{V}[\![x_1, \texttt{(dw}\ x_2\ e_1\ e_2\ e_3\texttt{)}]\!]$	if $\mathscr{V}[\![x_1, e_1]\!]$ or $\mathscr{V}[\![x_1, e_2]\!]$ or $\mathscr{V}[\![x_1, e_3]\!]$

Figure A.9b: Variable-assignment relation

paring two procedures is covered in section A.12.

A.9. Procedures and application

In evaluating a procedure call, the report leaves unspecified the order in which arguments are evaluated. So, our reduction system allows multiple, different reductions to occur, one for each possible order of evaluation.

To capture unspecified evaluation order but allow only evaluation that is consistent with some sequential ordering of the evaluation of an application's subexpressions, we use non-deterministic choice to first pick a subexpression to reduce only when we have not already committed to reducing some other subexpression. To achieve that effect, we limit the evaluation of application expressions to only those that have a single expression that is not fully reduced, as shown in the non-terminal F, in figure A.2b. To evaluate application expressions that have more than two arguments to evaluate, the rule [6mark] picks one of the subexpressions of an application that is not fully simplified and lifts it out in its own application, allowing it to be evaluated. Once one of the lifted expressions is evaluated, the [6appN] substitutes its value back into the original application.

The [6appN] rule also handles other applications whose arguments are finished by substituting the first argument for the first formal parameter in the expression. Its side-condition uses the relation in figure A.9b to ensure that there are no `set!` expressions with the parameter x_1 as a target. If there is such an assignment, the [6appN!] rule applies (see also section A.3 for a description of "fresh"). Instead of directly substituting the actual parameter for the formal parameter, it creates a new location in the store, initially bound the actual parameter, and substitutes a variable standing for that location in place of the formal parameter. The store, then, handles any eventual assignment to the parameter. Once all of the parameters have been substituted away, the rule [6app0] applies and evaluation of the body of the procedure begins.

At first glance, the rule [6appN] appears superfluous, since it seems like the rules could just reduce first by [6appN!] and then look up the variable when it is evaluated. There are two reasons why we keep the [6appN], however. The first is purely conventional: reducing applications via substitution is taught to us at an early age and is commonly used in rewriting systems in the literature. The second reason is more technical: the [6mark] rule requires that [6appN] be applied once e_i has been reduced to a value. [6appN!] would lift the value into the store and put a variable reference into the application, leading to another use of [6mark], and another use of [6appN!], which continues forever.

The rule [6μapp] handles a well-formed application of a function with a dotted parameter lists. It such an application into an application of an ordinary procedure by constructing a list of the extra arguments. Similarly, the rule [6μapp1] handles an application of a procedure that has a single variable as its parameter list.

The rule [6var] handles variable lookup in the store and [6set] handles variable assignment.

The next two rules [6proct] and [6procf] handle applications of `procedure?`, and the remaining rules cover applications of non-procedures and arity violations.

The rules in figure A.9c cover `apply`. The first rule, [6applyf], covers the case where the last argument to `apply` is the empty list, and simply reduces by erasing the empty

P_1[`(apply` $proc_1$ v_1 $\cdots$ `null)`]$\rightarrow$ [6applyf]
P_1[`(`$proc_1$ v_1 $\cdots$`)`]

`(store (`sf_1 $\cdots$ `(`pp_1 `(cons` v_2 v_3`))` sf_2 $\cdots$`)` E_1[`(apply` $proc_1$ v_1 $\cdots$ pp_1`)`]`)`$\rightarrow$ [6applyc]
`(store (`sf_1 $\cdots$ `(`pp_1 `(cons` v_2 v_3`))` sf_2 $\cdots$`)` E_1[`(apply` $proc_1$ v_1 $\cdots$ v_2 v_3`)`]`)`
$(\neg\mathscr{C}[\![pp_1, v_3,$ `(`sf_1 $\cdots$ `(`pp_1 `(cons` v_2 v_3`))` sf_2 $\cdots$`)`$]\!])$

`(store (`sf_1 $\cdots$ `(`pp_1 `(cons` v_2 v_3`))` sf_2 $\cdots$`)` E_1[`(apply` $proc_1$ v_1 $\cdots$ pp_1`)`]`)`$\rightarrow$ [6applyce]
`(store (`sf_1 $\cdots$ `(`pp_1 `(cons` v_2 v_3`))` sf_2 $\cdots$`)` E_1[`(raise (make-cond` "apply called on circular list"`))`]`)`
$(\mathscr{C}[\![pp_1, v_3,$ `(`sf_1 $\cdots$ `(`pp_1 `(cons` v_2 v_3`))` sf_2 $\cdots$`)`$]\!])$

P_1[`(apply` $nonproc$ v $\cdots$`)`]$\rightarrow$ [6applynf]
P_1[`(raise (make-cond` "can't apply non-procedure"`))`]

P_1[`(apply` $proc$ v_1 $\cdots$ v_2`)`]$\rightarrow$ [6applye]
P_1[`(raise (make-cond` "apply's last argument non-list"`))`] $(v_2 \notin list\text{-}v)$

P_1[`(apply)`]$\rightarrow$ [6apparity0]
P_1[`(raise (make-cond` "arity mismatch"`))`]

P_1[`(apply` v`)`]$\rightarrow$ [6apparity1]
P_1[`(raise (make-cond` "arity mismatch"`))`]

$\mathscr{C} \in 2^{pp \times val \times (sf\ \cdots)}$
$\mathscr{C}[\![pp_1, pp_2,$ `(`sf_1 $\cdots$ `(`pp_2 `(cons` v_1 v_2`))` sf_2 $\cdots$`)`$]\!]$ if $pp_1 = v_2$
$\mathscr{C}[\![pp_1, pp_2,$ `(`sf_1 $\cdots$ `(`pp_2 `(cons` v_1 v_2`))` sf_2 $\cdots$`)`$]\!]$ if $\mathscr{C}[\![pp_1, v_2,$ `(`sf_1 $\cdots$ `(`pp_2 `(cons` v_1 v_2`))` sf_2 $\cdots$`)`$]\!]$ and $pp_1 \neq v_2$

Figure A.9c: Apply

list and the `apply`. The second rule, [6applyc] covers a well-formed application of `apply` where `apply`'s final argument is a pair. It reduces by extracting the components of the pair from the store and putting them into the application of `apply`. Repeated application of this rule thus extracts all of the list elements passed to `apply` out of the store.

The remaining five rules cover the various violations that can occur when using `apply`. The first one covers the case where `apply` is supplied with a cyclic list. The next four cover applying a non-procedure, passing a non-list as the last argument, and supplying too few arguments to `apply`.

A.10. Call/cc and dynamic wind

The specification of `dynamic-wind` uses `(dw` x e e e`)` expressions to record which dynamic-wind *thunk*s are active at each point in the computation. Its first argument is an identifier that is globally unique and serves to identify invocations of `dynamic-wind`, in order to avoid exiting and re-entering the same dynamic context during a continuation switch. The second, third, and fourth arguments are calls to some *before*, *thunk*, and *after* procedures from a call to `dynamic-wind`. Evaluation only occurs in the middle expression; the `dw` expression only serves to record which *before* and *after* procedures need to be run during a continuation switch. Accordingly, the reduction rule for an application of `dynamic-wind` reduces to a call to the *before* procedure, a `dw` expression and a call to the *after* procedure, as shown in rule [6wind] in figure A.10. The next two rules cover abuses of the `dynamic-wind` procedure: calling it with non-procedures, and calling it with the wrong number of arguments. The [6dwdone] rule erases a `dw` expression when its second argument has finished evaluating.

The next two rules cover `call/cc`. The rule [6call/cc] creates a new continuation. It takes the context of the `call/cc` expression and packages it up into a `throw` expression that represents the continuation. The `throw` expression uses the fresh variable x to record where the application of `call/cc` occurred in the context for use in the [6throw] rule when the continuation is applied. That rule takes the arguments of the continuation, wraps them with a call to `values`, and puts them back into the place where the original call to `call/cc` occurred, replacing the current context with the context returned by the $\mathscr{T}$ metafunction.

The $\mathscr{T}$ (for "trim") metafunction accepts two D contexts and builds a context that matches its second argument, the destination context, except that additional calls to the *before* and *after* procedures from `dw` expressions in the context have been added.

The first clause of the $\mathscr{T}$ metafunction exploits the H context, a context that contains everything except `dw` expressions. It ensures that shared parts of the `dynamic-wind` context are ignored, recurring deeper into the two expression contexts as long as the first `dw` expression in each have

P_1[`(dynamic-wind` $proc_1$ $proc_2$ $proc_3$`)`]$\rightarrow$ [6wind]
P_1[`(begin (`$proc_1$`) (begin0 (dw` x `(`$proc_1$`) (`$proc_2$`) (`$proc_3$`)) (`$proc_3$`)))`] (x fresh)

P_1[`(dynamic-wind` v_1 v_2 v_3`)`]$\rightarrow$ [6winde]
P_1[`(raise (make-cond` "dynamic-wind expects procs"`))`] ($v_1 \notin proc$ or $v_2 \notin proc$ or $v_3 \notin proc$)

P_1[`(dynamic-wind` v_1 $\cdots$`)`]$\rightarrow$ [6dwarity]
P_1[`(raise (make-cond` "arity mismatch"`))`] ($\#v_1 \neq 3$)

P_1[`(dw` x e_1 `(values` v_1 $\cdots$`)` e_2`)`]$\rightarrow$ [6dwdone]
P_1[`(values` v_1 $\cdots$`)`]

`(store (`sf_1 $\cdots$`)` E_1[`(call/cc` v_1`)`]`)`$\rightarrow$ [6call/cc]
`(store (`sf_1 $\cdots$`)` E_1[`(`v_1 `(throw` x $E_1[x]$`))`]`)` (x fresh)

`(store (`sf_1 $\cdots$`)` E_1[`((throw` x_1 $E_2[x_1]$`)` v_1 $\cdots$`)`]`)`$\rightarrow$ [6throw]
`(store (`sf_1 $\cdots$`)` $\mathscr{T}[\![E_1, E_2]\!]$[`(values` v_1 $\cdots$`)`]`)`

$\mathscr{T} : E \times E \rightarrow E$
$\mathscr{T}[\![H_1[(\texttt{dw}\ x_1\ e_1\ E_1\ e_2)], H_2[(\texttt{dw}\ x_1\ e_3\ E_2\ e_4)]]\!] = H_2[(\texttt{dw}\ x_1\ e_3\ \mathscr{T}[\![E_1, E_2]\!]\ e_4)]$
$\mathscr{T}[\![E_1, E_2]\!] = (\texttt{begin}\ \mathscr{S}[\![E_1]\!][1]\ \mathscr{R}[\![E_2]\!])$ (otherwise)

$\mathscr{R} : E \rightarrow E$
$\mathscr{R}[\![H_1[(\texttt{dw}\ x_1\ e_1\ E_1\ e_2)]]\!] = H_1[(\texttt{begin}\ e_1\ (\texttt{dw}\ x_1\ e_1\ \mathscr{R}[\![E_1]\!]\ e_2))]$
$\mathscr{R}[\![H_1]\!] = H_1$ (otherwise)

$\mathscr{S} : E \rightarrow E$
$\mathscr{S}[\![E_1[(\texttt{dw}\ x_1\ e_1\ H_2\ e_2)]]\!] = \mathscr{S}[\![E_1]\!][(\texttt{begin0}\ (\texttt{dw}\ x_1\ e_1\ [\,]\ e_2)\ e_2)]$
$\mathscr{S}[\![H_1]\!] = [\,]$ (otherwise)

Figure A.10: Call/cc and dynamic wind

matching identifiers (x_1). The final rule is a catchall; it only applies when all the others fail and thus applies either when there are no `dw`s in the context, or when the `dw` expressions do not match. It calls the two other metafunctions defined in figure A.10 and puts their results together into a `begin` expression.

The $\mathscr{R}$ metafunction extracts all of the *before* procedures from its argument and the $\mathscr{S}$ metafunction extracts all of the *after* procedures from its argument. They each construct new contexts and exploit H to work through their arguments, one `dw` at a time. In each case, the metafunctions are careful to keep the right `dw` context around each of the procedures in case a continuation jump occurs during one of their evaluations. Since $\mathscr{R}$, receives the destination context, it keeps the intermediate parts of the context in its result. In contrast $\mathscr{S}$ discards all of the context except the `dw`s, since that was the context where the call to the continuation occurred.

A.11. Letrec

Figre A.11 shows the rules that handle `letrec` and `letrec*` and the supplementary expressions that they produce, `l!` and `reinit`. As a first approximation, both `letrec` and `letrec*` reduce by allocating locations in the store to hold the values of the init expressions, initializing those locations to `bh` (for "black hole"), evaluating the init expressions, and then using `l!` to update the locations in the store with the value of the init expressions. They also use `reinit` to detect when an init expression in a letrec is reentered via a continuation.

Before considering how `letrec` and `letrec*` use `l!` and `reinit`, first consider how `l!` and `reinit` behave. The first two rules in figure A.11 cover `l!`. It behaves very much like `set!`, but it initializes both ordinary variables, and variables that are current bound to the black hole (`bh`).

The next two rules cover ordinary `set!` when applied to a variable that is currently bound to a black hole. This situation can arise when the program assigns to a variable before letrec initializes it, eg `(letrec ((x (set! x 5))) x)`. The report specifies that either an implementation should perform the assignment, as reflected in the [6setdt] rule or it raise an exception, as reflected in the [6setdte] rule.

The [6dt] rule covers the case where a variable is referred to before the value of a init expression is filled in, which

(store (sf_1 $\cdots$ (x_1 bh) sf_2 $\cdots$) E_1[(l! x_1 v_2)])$\rightarrow$ [6initdt]
(store (sf_1 $\cdots$ (x_1 v_2) sf_2 $\cdots$) E_1[unspecified])

(store (sf_1 $\cdots$ (x_1 v_1) sf_2 $\cdots$) E_1[(l! x_1 v_2)])$\rightarrow$ [6initv]
(store (sf_1 $\cdots$ (x_1 v_2) sf_2 $\cdots$) E_1[unspecified])

(store (sf_1 $\cdots$ (x_1 bh) sf_2 $\cdots$) E_1[(set! x_1 v_1)])$\rightarrow$ [6setdt]
(store (sf_1 $\cdots$ (x_1 v_1) sf_2 $\cdots$) E_1[unspecified])

(store (sf_1 $\cdots$ (x_1 bh) sf_2 $\cdots$) E_1[(set! x_1 v_1)])$\rightarrow$ [6setdte]
(store (sf_1 $\cdots$ (x_1 bh) sf_2 $\cdots$) E_1[(raise (make-cond "letrec variable touched"))])

(store (sf_1 $\cdots$ (x_1 bh) sf_2 $\cdots$) $E_1[x_1]$)$\rightarrow$ [6dt]
(store (sf_1 $\cdots$ (x_1 bh) sf_2 $\cdots$) E_1[(raise (make-cond "letrec variable touched"))])

(store (sf_1 $\cdots$ (x_1 #f) sf_2 $\cdots$) E_1[(reinit x_1)])$\rightarrow$ [6init]
(store (sf_1 $\cdots$ (x_1 #t) sf_2 $\cdots$) E_1['*ignore*])

(store (sf_1 $\cdots$ (x_1 #t) sf_2 $\cdots$) E_1[(reinit x_1)])$\rightarrow$ [6reinit]
(store (sf_1 $\cdots$ (x_1 #t) sf_2 $\cdots$) E_1['*ignore*])

(store (sf_1 $\cdots$ (x_1 #t) sf_2 $\cdots$) E_1[(reinit x_1)])$\rightarrow$ [6reinite]
(store (sf_1 $\cdots$ (x_1 #t) sf_2 $\cdots$) E_1[(raise (make-cond "reinvoked continuation of letrec init"))])

(store (sf_1 $\cdots$) E_1[(letrec ((x_1 e_1) $\cdots$) e_2 e_3 $\cdots$)])$\rightarrow$ [6letrec]
(store (sf_1 $\cdots$ (lx bh) $\cdots$ (ri #f) $\cdots$)
  E_1[((lambda (x_1 $\cdots$) (l! lx x_1) $\cdots$ $\{x_1 \mapsto lx \cdots\}e_2$ $\{x_1 \mapsto lx \cdots\}e_3$ $\cdots$)
    (begin0 $\{x_1 \mapsto lx \cdots\}e_1$ (reinit ri)) $\cdots$)])
  ($lx \cdots$ fresh, $ri \cdots$ fresh)

(store (sf_1 $\cdots$) E_1[(letrec* ((x_1 e_1) $\cdots$) e_2 e_3 $\cdots$)])$\rightarrow$ [6letrec*]
(store (sf_1 $\cdots$ (lx bh) $\cdots$ (ri #f) $\cdots$) E_1[$\{x_1 \mapsto lx \cdots\}$(begin (begin (l! lx e_1) (reinit ri)) $\cdots$ e_2 e_3 $\cdots$)])
  ($lx \cdots$ fresh, $ri \cdots$ fresh)

Figure A.11: Letrec and letrec*

must always raise an exception.

A `reinit` expression is used to detect a program that captures a continuation in an initialization expression and returns to it, as shown in the three rules [6init], [6reinit], and [6reinite]. The `reinit` form accepts an identifier that is bound in the store to a boolean as its argument. Those are identifiers are initially `#f`. When `reinit` is evaluated, it checks the value of the variable and, if it is still `#f`, it changes it to `#t`. If it is already `#t`, then `reinit` either just does nothing, or it raises an exception, in keeping with the two legal behaviors of `letrec` and `letrec*`.

The last two rules in figure A.11 put together `l!` and `reinit`. The [6letrec] rule reduces a `letrec` expression to an application expression, in order to capture the unspecified order of evaluation of the init expressions. Each init expression is wrapped in a `begin0` that records the value of the init and then uses `reinit` to detect continuations that return to the init expression. Once all of the init expressions have been evaluated, the procedure on the right-hand side of the rule is invoked, causing the value of the init expression to be filled in the store, and evaluation continues with the body of the original `letrec` expression.

The [6letrec*] rule behaves similarly, but uses a `begin` expression rather than an application, since the init expressions are evaluated from left to right. Moreover, each init expression is filled into the store as it is evaluated, so that subsequent init expressions can refer to its value.

A.12. Underspecification

The rules in figure A.12 cover aspects of the semantics that are explicitly unspecified. Implementations can replace the rules [6ueqv], [6uval] and with different rules that cover the left-hand sides and, as long as they follow the informal specification, any replacement is valid. Those three situations correspond to the case when `eqv?` applied to two procedures and when multiple values are used in a single-value context.

The remaining rules in figure A.12 cover the results from the assignment operations, `set!`, `set-car!`, and `set-cdr!`. An implementation does not adjust those rules, but instead renders them useless by adjusting the rules that insert `unspecified`: [6setcar], [6setcdr], [6set], and [6setd]. Those

$P[$`(eqv?` $proc_1$ $proc_2$`)`$]$	$\rightarrow$ **unknown:** equivalence of procedures	[6ueqv]
$P[$`(values` v_1 $\cdots$`)`$]_\circ$	$\rightarrow$ **unknown:** context expected one value, received $\#v_1$ $(\#v_1 \neq 1)$	[6uval]
$P[U[$`unspecified`$]]$	$\rightarrow$ **unknown:** unspecified result	[6udemand]
`(store (`sf $\cdots$`) unspecified)`	$\rightarrow$ **unknown:** unspecified result	[6udemandtl]
$P_1[$`(begin unspecified` e_1 e_2 $\cdots$`)`$]$	$\rightarrow P_1[$`(begin` e_1 e_2 $\cdots$`)`$]$	[6ubegin]
$P_1[$`(handlers` v $\cdots$ `unspecified)`$]$	$\rightarrow P_1[$`unspecified`$]$	[6uhandlers]
$P_1[$`(dw` x e_1 `unspecified` e_2`)`$]$	$\rightarrow P_1[$`unspecified`$]$	[6udw]
$P_1[$`(begin0 (values` v_1 $\cdots$`) unspecified` e_1 $\cdots$`)`$]$	$\rightarrow P_1[$`(begin0 (values` v_1 $\cdots$`)` e_1 $\cdots$`)`$]$	[6ubegin0]
$P_1[$`(begin0 unspecified (values` v_2 $\cdots$`)` e_2 $\cdots$`)`$]$	$\rightarrow P_1[$`(begin0 unspecified` e_2 $\cdots$`)`$]$	[6ubegin0u]
$P_1[$`(begin0 unspecified unspecified` e_2 $\cdots$`)`$]$	$\rightarrow P_1[$`(begin0 unspecified` e_2 $\cdots$`)`$]$	[6ubegin0uu]

Figure A.12: Explicitly unspecified behavior

rules can be adjusted by replacing `unspecified` with any number of values in those rules.

So, the remaining rules just specify the minimal behavior that we know that a value or values must have and otherwise reduce to an **unknown:** state. The rule [6udemand] drops `unspecified` in the `U` context. See figure A.2b for the precise definition of `U`, but intuitively it is a context that is only a single expression layer deep that contains expressions whose value depends on the value of their subexpressions, like the first subexpression of a `if`. Following that are rules that discard `unspecified` in expressions that discard the results of some of their subexpressions. The [6ubegin] shows how `begin` discards its first expression when there are more expressions to evaluate. The next two rules, [6uhandlers] and [6udw] propagate `unspecified` to their context, since they also return any number of values to their context. Finally, the two `begin0` rules preserve `unspecified` until the rule [6begin01] can return it to its context.

Appendix B. Sample definitions for derived forms

This appendix contains sample definitions for some of the keywords described in this report in terms of simpler forms:

`cond`

The `cond` keyword (section 11.4.5) could be defined in terms of `if`, `let` and `begin` using `syntax-rules` as follows:

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))
```

`case`

The `case` keyword (section 11.4.5) could be defined in terms of `let`, `cond`, and `memv` (see library chapter 3) using `syntax-rules` as follows:

```
(define-syntax case
  (syntax-rules (else)
    ((case expr0
       ((key ...) res1 res2 ...)
       ...
       (else else-res1 else-res2 ...))
     (let ((tmp expr0))
```

```
        (cond
          ((memv tmp '(key ...)) res1 res2 ...)
          ...
          (else else-res1 else-res2 ...))))
    ((case expr0
       ((keya ...) res1a res2a ...)
       ((keyb ...) res1b res2b ...)
       ...)
     (let ((tmp expr0))
       (cond
         ((memv tmp '(keya ...)) res1a res2a ...)
         ((memv tmp '(keyb ...)) res1b res2b ...)
         ...)))))
```

let*

The `let*` keyword (section 11.4.6) could be defined in terms of `let` using `syntax-rules` as follows:

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

letrec

The `letrec` keyword (section 11.4.6) could be defined approximately in terms of `let` and `set!` using `syntax-rules`, using a helper to generate the temporary variables needed to hold the values before the assignments are made, as follows:

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec () body1 body2 ...)
     (let () body1 body2 ...))
    ((letrec ((var init) ...) body1 body2 ...)
     (letrec-helper
       (var ...)
       ()
       ((var init) ...)
       body1 body2 ...))))

(define-syntax letrec-helper
  (syntax-rules ()
    ((letrec-helper
       ()
       (temp ...)
       ((var init) ...)
       body1 body2 ...)
     (let ((var <undefined>) ...)
       (let ((temp init) ...)
         (set! var temp)
         ...)
         (let () body1 body2 ...)))
    ((letrec-helper
       (x y ...)
       (temp ...)
       ((var init) ...)
       body1 body2 ...)
     (letrec-helper
       (y ...)
       (newtemp temp ...)
       ((var init) ...)
       body1 body2 ...))))
```

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

A simpler definition using `syntax-case` and `generate-temporaries` is given in library chapter 12.

letrec*

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

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

The syntax `<undefined>` is as in the definition of `letrec` above.

let-values

The following definition of `let-values` (section 11.4.6) using `syntax-rules` employs a pair of helpers to create temporary names for the formals.

```
(define-syntax let-values
  (syntax-rules ()
    ((let-values (binding ...) body1 body2 ...)
     (let-values-helper1
       ()
       (binding ...)
       body1 body2 ...))))

(define-syntax let-values-helper1
  ;; map over the bindings
  (syntax-rules ()
    ((let-values
       ((id temp) ...)
       ()
       body1 body2 ...)
     (let ((id temp) ...) body1 body2 ...))
```

```
    ((let-values
       assocs
       ((formals1 expr1) (formals2 expr2) ...)
       body1 body2 ...)
     (let-values-helper2
       formals1
       ()
       expr1
       assocs
       ((formals2 expr2) ...)
       body1 body2 ...))))

(define-syntax let-values-helper2
  ;; create temporaries for the formals
  (syntax-rules ()
    ((let-values-helper2
       ()
       temp-formals
       expr1
       assocs
       bindings
       body1 body2 ...)
     (call-with-values
       (lambda () expr1)
       (lambda temp-formals
         (let-values-helper1
           assocs
           bindings
           body1 body2 ...))))
    ((let-values-helper2
       (first . rest)
       (temp ...)
       expr1
       (assoc ...)
       bindings
       body1 body2 ...)
     (let-values-helper2
       rest
       (temp ... newtemp)
       expr1
       (assoc ... (first newtemp))
       bindings
       body1 body2 ...))
    ((let-values-helper2
       rest-formal
       (temp ...)
       expr1
       (assoc ...)
       bindings
       body1 body2 ...)
     (call-with-values
       (lambda () expr1)
       (lambda (temp ... . newtemp)
         (let-values-helper1
           (assoc ... (rest-formal newtemp))
           bindings
           body1 body2 ...))))))
```

`let*-values`

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

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

`let`

The `let` keyword could be defined in terms of `lambda` and `letrec` using `syntax-rules` as follows:

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
        tag)
      val ...))))
```

Appendix C. Additional material

This report itself, as well as more material related to this report such as reference implementations of some parts of Scheme and archives of mailing lists discussing this report is at

`http://www.r6rs.org/`

The Schemers web site at

`http://www.schemers.org/`

as well as the Readscheme site at

`http://library.readscheme.org/`

contain extensive Scheme bibliographies, as well as papers, programs, implementations, and other material related to Scheme.

Appendix D. Example

This section describes an example consisting of the `(runge-kutta)` library, which provides an `integrate-system` procedure that integrates the system

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

As the `(runge-kutta)` library makes use of the `(rnrs base (6))` library, its skeleton is as follows:

```
#!r6rs
(library (runge-kutta)
  (export integrate-system
          head tail)
  (import (rnrs base))
  ⟨library body⟩)
```

The procedure definitions described below go in the place of ⟨library body⟩.

The parameter `system-derivative` is a function that takes a system state (a vector of values for the state variables $y_1, \ldots, y_n$) and produces a system derivative (the values $y'_1, \ldots, y'_n$). The parameter `initial-state` provides an initial system state, and `h` is an initial guess for the length of the integration step.

The value returned by `integrate-system` is an infinite stream of system states.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (lambda ()
                        (map-streams next states)))))
        states))))
```

The `runge-kutta-4` procedure takes a function, `f`, that produces a system derivative from a system state. The `runge-kutta-4` procedure produces a function that takes a system state and produces a new system state.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3))))))))

(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

The `map-streams` procedure is analogous to `map`: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (lambda () (map-streams f (tail s))))))
```

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a procedure that delivers the rest of the stream.

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

The following program illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
#!r6rs
(import (rnrs base)
        (rnrs io simple)
        (runge-kutta))

(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
```

```
    (damped-oscillator 10000 1000 .001)
    '#(1 0)
    .01))

(letrec ((loop (lambda (s)
                 (newline)
                 (write (head s))
                 (loop (tail s)))))
  (loop the-states))

(close-output-port (current-output-port))
```

This prints output like the following:

```
#(1 0)
#(0.99895054 9.994835e-6)
#(0.99780226 1.9978681e-5)
#(0.9965554 2.9950552e-5)
#(0.9952102 3.990946e-5)
#(0.99376684 4.985443e-5)
#(0.99222565 5.9784474e-5)
#(0.9905868 6.969862e-5)
#(0.9888506 7.9595884e-5)
#(0.9870173 8.94753e-5)
```

Appendix E. Language changes

This chapter describes most of the changes that have been made to Scheme since the "Revised[5] Report" [14] was published:

- Scheme source code now uses the Unicode character set. Specifically, the character set that can be used for identifiers has been greatly expanded.
- Identifiers can now start with the characters `->`.
- Identifiers and symbol literals are now case-sensitive.
- Identifiers and representations of characters, booleans, number objects, and `.` must be explicitly delimited.
- `#` is now a delimiter.
- Bytevector literal syntax has been added.
- Matched square brackets can be used synonymously with parentheses.
- The read-syntax abbreviations `#'` (for `syntax`), `` #` `` (for `quasisyntax`), `#,` (for `unsyntax`), and `#,@` (for `unsyntax-splicing` have been added; see section 4.3.5.)
- `#` can no longer be used in place of digits in number representations.
- The external representation of number objects can now include a mantissa width.
- Literals for NaNs and infinities were added.
- String and character literals can now use a variety of escape sequences.
- Block and datum comments have been added.
- The `#!r6rs` comment for marking report-compliant lexical syntax has been added.
- Characters are now specified to correspond to Unicode scalar values.
- Many of the procedures and syntactic forms of the language are now part of the `(rnrs base (6))` library. Some procedures and syntactic forms have been moved to other libraries; see figure A.1.
- The base language has the following new procedures and syntactic forms: `letrec*`, `let-values`, `let*-values`, `real-valued?`, `rational-valued?`, `integer-valued?`, `exact`, `inexact`, `finite?`, `infinite?`, `nan?`, `div`, `mod`, `div-and-mod`, `div0`, `mod0`, `div0-and-mod0`, `exact-integer-sqrt`, `boolean=?`, `symbol=?`, `string-for-each`, `vector-map`, `vector-for-each`, `error`, `assertion-violation`, `assert`, `call/cc`, `identifier-syntax`.
- The following procedures have been removed: `char-ready?`, `transcript-on`, `transcript-off`, `load`.
- The case-insensitive string comparisons (`string-ci=?`, `string-ci<?`, `string-ci>?`, `string-ci<=?`, `string-ci>=?`) operate on the case-folded versions of the strings rather than as the simple lexicographic ordering induced by the corresponding character comparison procedures.
- Libraries have been added to the language.
- A number of standard libraries are described in a separate report [24].
- Many situations that "were an error" now have defined or constrained behavior. In particular, many are now specified in terms of the exception system.
- The full numerical tower is now required.
- The semantics for the transcendental functions has been specified more fully.
- The semantics of `expt` for zero bases has been refined.
- In `syntax-rules` forms, a `_` may be used in place of the keyword.
- The `let-syntax` and `letrec-syntax` no longer introduce a new environment for their bodies.

identifier	moved to	identifier	moved to
`assoc`	`(rnrs lists (6))`	`inexact->exact`	`(rnrs r5rs (6))`
`assv`	`(rnrs lists (6))`	`member`	`(rnrs lists (6))`
`assq`	`(rnrs lists (6))`	`memv`	`(rnrs lists (6))`
`call-with-input-file`	`(rnrs io simple (6))`	`memq`	`(rnrs lists (6))`
`call-with-output-file`	`(rnrs io simple (6))`	`modulo`	`(rnrs r5rs (6))`
`char-upcase`	`(rnrs unicode (6))`	`newline`	`(rnrs io simple (6))`
`char-downcase`	`(rnrs unicode (6))`	`null-environment`	`(rnrs r5rs (6))`
`char-ci=?`	`(rnrs unicode (6))`	`open-input-file`	`(rnrs io simple (6))`
`char-ci<?`	`(rnrs unicode (6))`	`open-output-file`	`(rnrs io simple (6))`
`char-ci>?`	`(rnrs unicode (6))`	`peek-char`	`(rnrs io simple (6))`
`char-ci<=?`	`(rnrs unicode (6))`	`quotient`	`(rnrs r5rs (6))`
`char-ci>=?`	`(rnrs unicode (6))`	`read`	`(rnrs io simple (6))`
`char-alphabetic?`	`(rnrs unicode (6))`	`read-char`	`(rnrs io simple (6))`
`char-numeric?`	`(rnrs unicode (6))`	`remainder`	`(rnrs r5rs (6))`
`char-whitespace?`	`(rnrs unicode (6))`	`scheme-report-environment`	`(rnrs r5rs (6))`
`char-upper-case?`	`(rnrs unicode (6))`	`set-car!`	`(rnrs mutable-pairs (6))`
`char-lower-case?`	`(rnrs unicode (6))`	`set-cdr!`	`(rnrs mutable-pairs (6))`
`close-input-port`	`(rnrs io simple (6))`	`string-ci=?`	`(rnrs unicode (6))`
`close-output-port`	`(rnrs io simple (6))`	`string-ci<?`	`(rnrs unicode (6))`
`current-input-port`	`(rnrs io simple (6))`	`string-ci>?`	`(rnrs unicode (6))`
`current-output-port`	`(rnrs io simple (6))`	`string-ci<=?`	`(rnrs unicode (6))`
`display`	`(rnrs io simple (6))`	`string-ci>=?`	`(rnrs unicode (6))`
`do`	`(rnrs control (6))`	`string-set!`	`(rnrs mutable-strings (6))`
`eof-object?`	`(rnrs io simple (6))`	`string-fill!`	`(rnrs mutable-strings (6))`
`eval`	`(rnrs eval (6))`	`with-input-from-file`	`(rnrs io simple (6))`
`delay`	`(rnrs r5rs (6))`	`with-output-to-file`	`(rnrs io simple (6))`
`exact->inexact`	`(rnrs r5rs (6))`	`write`	`(rnrs io simple (6))`
`force`	`(rnrs r5rs (6))`	`write-char`	`(rnrs io simple (6))`

Figure A.1: Identifiers moved to libraries

- For implementations that support NaNs or infinities, many arithmetic operations have been specified on these values consistently with IEEE 754.

- For implementations that support a distinct -0.0, the semantics of many arithmetic operations with regard to -0.0 has been specified consistently with IEEE 754.

- Scheme's real number objects now have an exact zero as their imaginary part.

- The specification of `quasiquote` has been extended. Nested quasiquotations work correctly now, and `unquote` and `unquote-splicing` have been extended to several operands.

- Procedures now may or may not refer to locations. Consequently, `eqv?` is now unspecified in a few cases where it was specified before.

- The mutability of the values of `quasiquote` structures has been specified to some degree.

- The dynamic environment of the *before* and *after* procedures of `dynamic-wind` is now specified.

- Various expressions that have only side effects are now allowed to return an arbitrary number of values.

- The order and semantics for macro expansion has been more fully specified.

- Internal definitions are now defined in terms of `letrec*`.

- The old notion of program structure and Scheme's top-level environment has been replaced by top-level programs and libraries.

- The denotational semantics has been replaced by an operational semantics based on an earlier semantics for the language of the "Revised5 Report" [14, 18].

REFERENCES

[1] J. W. Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy P. Naur, A. J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *ACM Conference on Lisp and Functional Programming*, pages 86–95, Snowbird, Utah, 1988. ACM Press.

[3] Scott Bradner. Key words for use in RFCs to indicate requirement levels. `http://www.ietf.org/rfc/rfc2119.txt`, March 1997. RFC 2119.

[4] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116, Philadelphia, PA, USA, May 1996. ACM Press.

[5] William Clinger. Proper tail recursion and space efficiency. In Keith Cooper, editor, *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, pages 174–185, Montreal, Canada, June 1998. ACM Press. Volume 33(5) of SIGPLAN Notices.

[6] William Clinger and Jonathan Rees. Macros that work. In *Proc. 1991 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 155–162, Orlando, Florida, January 1991. ACM Press.

[7] William D. Clinger. How to read floating point numbers accurately. In *Proc. Conference on Programming Language Design and Implementation '90*, pages 92–101, White Plains, New York, USA, June 1990. ACM.

[8] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005. `http://www.scheme.com/csug7/`.

[9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[10] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. `http://www.cs.utah.edu/plt/publications/pllc.pdf`, 2003.

[11] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, July 2006. `http://download.plt-scheme.org/doc/352/html/mzscheme/`.

[12] Daniel P. Friedman, Christopher Haynes, Eugene Kohlbecker, and Mitchell Wand. *Scheme 84 interim reference manual*. Indiana University, January 1985. Indiana University Computer Science Technical Report 153.

[13] IEEE standard 754-1985. IEEE standard for binary floating-point arithmetic, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[14] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[15] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[16] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, August 1986.

[17] Jacob Matthews and Robert Bruce Findler. An operational semantics for R5RS Scheme. In J. Michael Ashley and Michael Sperber, editors, *Proceedings of the Sixth Workshop on Scheme and Functional Programming*, pages 41–54, Tallin, Estonia, September 2005. Indiana University Technical Report TR619.

[18] Jacob Matthews and Robert Bruce Findler. An operational semantics for Scheme. *Journal of Functional Programming*, 2007. From `http://www.cambridge.org/journals/JFP/`.

[19] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. 15th Conference on Rewriting Techniques and Applications*, Aachen, June 2004. Springer-Verlag.

[20] MIT Department of Electrical Engineering and Computer Science. *Scheme manual, seventh edition*, September 1984.

[21] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. *The T manual*. Yale University Computer Science Department, fourth edition, January 1984.

[22] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised^6 report on the algorithmic language Scheme (Non-Normative appendices). `http://www.r6rs.org/`, 2007.

[23] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised^6 report on the algorithmic language Scheme (Rationale). `http://www.r6rs.org/`, 2007.

[24] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, and Jonathan Rees. Revised6 report on the algorithmic language Scheme (Libraries). `http://www.r6rs.org/`, 2007.

[25] Guy Lewis Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, MA, second edition, 1990.

[26] Texas Instruments, Inc. *TI Scheme Language Reference Manual*, November 1985. Preliminary version 1.0.

[27] The Unicode Consortium. The Unicode standard, version 5.0.0. defined by: *The Unicode Standard, Version 5.0* (Boston, MA, Addison-Wesley, 2007. ISBN 0-321-48091-0), 2007.

[28] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.

[29] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The index includes entries from the library document; the entries are marked with "(library)".