Revised^6 Report on the Algorithmic Language Scheme
— Rationale —

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*)

~~**26 September 2007**~~
****Unofficial version incorporating errata****
`https://standards.scheme.org/`
21 July 2019

SUMMARY

This document describes rationales for some of the design decisions behind the *Revised^6 Report on the Algorithmic Language Scheme*. The focus is on changes made since the last revision on the report. Moreover, numerous fundamental design decisions of Scheme are explained. This report also contains some historical notes. The formal comments submitted for drafts of the report and their responses, as archived on `http://www.r6rs.org/`, provides additional background information on many decisions that are reflected in the report.

This document is not intended to be an exhaustive justification for every decision and design aspect of the report. Instead, it provides information about some of the issues considered by the editors' committee when decisions were made, as background information and as guidelines for future decision makers. As such, the rationales given here may not be convincing to every reader, but they convinced the editors at the time the respective decisions were made.

This document frequently refers back to the *Revised^6 Report on the Algorithmic Language Scheme* [34], the *Revised^6 Report on the Algorithmic Language Scheme — Libraries —* [35], and the *Revised^6 Report on the Algorithmic Language Scheme — Non-Normative Appendices —* [33]; specific references to the report are identified by designations such as "report section" or "report chapter", references to the library report are identified by designations such as "library section" or "library chapter", and references to the appendices are identified by designations such as "appendix" or "appendix section". This document frequently refers to the whole *Revised^6 Report on the Algorithmic Language Scheme* as "R^{6}RS", and to the *Revised^5 Report on the Algorithmic Language Scheme* as "R^{5}RS".

CONTENTS

1 Historical background . . . 3
2 Requirement levels . . . 3
3 Numbers . . . 3
3.1 Infinities, NaNs . . . 3
3.2 Distinguished -0.0 . . . 4
4 Lexical syntax and datum syntax . . . 4
4.1 Symbol and identifier syntax . . . 4
4.2 Comments . . . 4
4.3 Future extensions . . . 5
5 Semantic concepts . . . 5
5.1 Argument and subform checking . . . 5
5.2 Safety . . . 5
5.3 Proper tail recursion . . . 5
6 Entry format . . . 6
7 Libraries . . . 6
7.1 Syntax . . . 6
7.2 Local import . . . 6
7.3 Local modules . . . 6
7.4 Fixed `import` and `export` clauses . . . 6
7.5 Instantiation and initialization . . . 7
7.6 Immutable exports . . . 7
7.7 Compound library names . . . 7
7.8 Versioning . . . 7
7.9 Treatment of different versions . . . 7
8 Top-level programs . . . 7
9 Primitive syntax . . . 8
9.1 Unspecified evaluation order . . . 8
10 Expansion process . . . 8
11 Base library . . . 8
11.1 Library organization . . . 8
11.2 Bodies . . . 8
11.3 Export levels . . . 9
11.4 Binding forms . . . 9
11.5 Equivalence predicates . . . 9
11.6 Arithmetic . . . 10
11.7 Characters and strings . . . 12
11.8 Symbols . . . 13
11.9 Control features . . . 13
11.10 Macro transformers . . . 13
12 Formal semantics . . . 14
13 Unicode . . . 14
13.1 Case mapping . . . 14
14 Bytevectors . . . 14
15 List utilities . . . 14
15.1 Notes on individual procedures . . . 14
16 Sorting . . . 14
17 Control structures . . . 15
17.1 `when` and `unless` . . . 15
17.2 `case-lambda` . . . 15
18 Records . . . 15
18.1 Syntactic layer . . . 15
18.2 Positional access and field names . . . 15
18.3 Lack of multiple inheritance . . . 15
18.4 Constructor mechanism . . . 15
18.5 Sealed record types . . . 15
19 Conditions and exceptions . . . 16
19.1 Exceptions . . . 16
19.2 Conditions . . . 16
20 I/O . . . 16
20.1 File names . . . 16
20.2 File options . . . 17
20.3 End-of-line styles . . . 17
20.4 Error-handling modes . . . 17
20.5 Binary and textual ports . . . 17
20.6 File positions . . . 17
20.7 Freshness of standard ports . . . 17
20.8 Argument conventions . . . 17
21 File system . . . 17
22 Arithmetic . . . 17
22.1 Fixnums and flonums . . . 17
22.2 Bitwise operations . . . 18
22.3 Notes on individual procedures . . . 18
23 `syntax-case` . . . 18
24 Hashtables . . . 18
24.1 Caching . . . 18
24.2 Immutable hashtables . . . 18
24.3 Hash functions . . . 18
25 Enumerations . . . 18
26 Composite library . . . 19
27 Mutable pairs . . . 19
28 Mutable strings . . . 19
References . . . 19

1. Historical background

The *Revised*^{6} *Report on the Algorithmic Language Scheme* (R^{6}RS for short) is the sixth of the Revised Reports on Scheme.

The first description of Scheme was written by Gerald Jay Sussman and Guy Lewis Steele Jr. in 1975 [39]. A revised report by Steele and Sussman [38] appeared in 1978 and described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [36]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [27, 25, 12]. An introductory computer science textbook using Scheme was published in 1984 [1]. A number of textbooks describing and using Scheme have been published since [8].

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Participating in this workshop were Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Daniel Friedman, Robert Halstead, Chris Hanson, Christopher Haynes, Eugene Kohlbecker, Don Oxley, Jonathan Rees, Guillermo Rozas, Gerald Jay Sussman, and Mitchell Wand. Their report [3], edited by Will Clinger, was published at MIT and Indiana University in the summer of 1985. Further revision took place in the spring of 1986 [4] (edited by Jonathan Rees and Will Clinger), and in the spring of 1988 [5] (also edited by Will Clinger and Jonathan Rees). Another revision published in 1998, edited by Richard Kelsey, Will Clinger and Jonathan Rees, reflected further revisions agreed upon in a meeting at Xerox PARC in June 1992 [21].

Attendees of the Scheme Workshop in Pittsburgh in October 2002 formed a Strategy Committee to discuss a process for producing new revisions of the report. The strategy committee drafted a charter for Scheme standardization. This charter, together with a process for selecting editorial committees for producing new revisions of the report, was confirmed by the attendees of the Scheme Workshop in Boston in November 2003. Subsequently, a Steering Committee according to the charter was selected, consisting of Alan Bawden, Guy L. Steele Jr., and Mitch Wand. An editors' committee charged with producing a new revision of the report was also formed at the end of 2003, consisting of Will Clinger, R. Kent Dybvig, Marc Feeley, Matthew Flatt, Richard Kelsey, Manuel Serrano, and Mike Sperber, with Marc Feeley acting as Editor-in-Chief. Richard Kelsey resigned from the committee in April 2005, and was replaced by Anton van Straaten. Marc Feeley and Manuel Serrano resigned from the committee in January 2006. Subsequently, the charter was revised to reduce the size of the editors' committee to five and to replace the office of Editor-in-Chief by a Chair and a Project Editor [29]. R. Kent Dybvig served as Chair, and Mike Sperber served as Project Editor. Will Clinger resigned from the committee in May 2007. Parts of the report were posted as Scheme Requests for Implementation (SRFIs, see `http://srfi.schemers.org/`) and discussed by the community before being revised and finalized for the report [14, 2, 6, 13, 9]. Jacob Matthews and Robby Findler wrote the operational semantics for the language core, based on an earlier semantics for the language of the "Revised^5 Report" [24].

2. Requirement levels

R^{6}RS distinguishes between different requirement levels, both for the programmer and for the implementation. Specifically, the distinction between "should" and "must" is important: For example, "should" is used for restrictions on argument types that are undecidable or potentially too expensive to enforce. The use of "should" allows implementations to perform quite extensive checking of restrictions on arguments (see section 5.1), but also to eschew more expensive checks.

3. Numbers

3.1. Infinities, NaNs

Infinities and NaNs are artifacts that help deal with the inexactness of binary floating-point arithmetic. The semantics dealing with infinities and NaNs, or the circumstances leading to their generation are somewhat arbitrary. However, as most Scheme implementation use an IEEE-754-conformant implementation [18] of flonums, R^{6}RS uses the particular semantics from this standard as the basis for the treatment of infinities and NaNs in the report. This is also the reason why infinities and NaNs are flonums and thus inexact real number objects, allowing Scheme systems to exploit the closure properties arising from their being part of the standard IEEE-754 floating-point representation. See section 11.6.6 for details on closure properties.

Infinities and NaNs are not considered integers (or even rational) by R^{6}RS. Despite this, the `ceiling`, `floor`, `round`, and `truncate` procedures (and their `fl`-prefixed counterparts) return an infinity or NaN when given an infinity or NaN as an argument. This has the advantage of allowing these procedures to take arbitrary real (or flonum) arguments but the disadvantage that they do not always return integer values. The integer values of the mathematical equivalents of these procedures are, in fact, infinite for infinite inputs. Also, while infinities are not considered integers, they might represent infinite integers. So the extension to infinities, at least, makes sense. The extension to NaNs is somewhat more arbitrary.

R^{6}RS intentionally does not require a Scheme implementation to use infinities and NaNs as specified in IEEE 754. Hence, support for them is optional.

3.2. Distinguished -0.0

A distinguished -0.0 is another artifact of IEEE 754, which can be used to construct certain branch cuts. A Scheme implementation is not required to distinguish -0.0. If it does, however, the behavior of the transcendental functions is sensitive to the distinction.

4. Lexical syntax and datum syntax

4.1. Symbol and identifier syntax

4.1.1. Escaped symbol constituents

While revising the syntax of symbols and identifiers, the editors' goal was to make symbols subject to write/read invariance, i.e. to allow each symbol to be written out using `put-datum` (section 8.2.12) or `write` (section 8.3), and read back in using `get-datum` (section 8.2.9) or `read` (section 8.3), yielding the same symbol. This was not the case in *Revised^5 Report on the Algorithmic Language Scheme*, as symbols could contain arbitrary characters such as spaces which could not be part of their external representation. Moreover, symbols could be distinguished internally by case, whereas their external representation could not.

For representing unusual characters in the symbol syntax, the report provides the `\x` escape syntax, which allows an arbitrary Unicode scalar value to be specified. This also has the advantage that arbitrary symbols can be represented using only ASCII, which allows referencing them from Scheme programs restricted to ASCII or some other subset of Unicode.

Among existing implementations of Scheme, a popular choice for extending the set of characters that can occur in symbols is the vertical-bar syntax of Common Lisp. The vertical-bar syntax of Common Lisp carries the risk of confusing the syntax of identifiers with that of consecutive lexemes, and also does not allow representing arbitrary characters using only ASCII. Consequently, it was not adopted for R^{6}RS.

4.1.2. Case sensitivity

The change from case-insensitive syntax in R^{5}RS to case-sensitive syntax is a major change. Many technical arguments exist in favor of both case sensitivity and case insensitivity, and any attempt to list them all here would be incomplete.

The editors decided to switch to case sensitivity, because they perceived that a significant majority of the Scheme community favored the change. This perception has been strengthened by polls at the 2004 Scheme workshop, on the `plt-scheme` mailing list, and the `r6rs-discuss` mailing list.

The suggested directives described in appendix B allow programs to specify that a section of the code (or other syntactic data) was written under the old assumption of case-insensitivity and therefore must be case-folded upon reading.

4.1.3. Identifiers starting with `->`

R^{6}RS introduces a special rule in the lexical syntax for identifiers starting with the characters `->`. In R^{5}RS, such identifiers are not valid lexemes. (In R^{5}RS, a lexeme starting with a `-` character—except for `-` itself—must be a representation of a number object.) However, many existing Scheme implementations prior to R^{6}RS already supported identifiers starting with `->`. (Many readers would classify any lexeme as an identifier starting with `-` for which `string->number` returns `#f`.) As a result, a significant amount of otherwise portable Scheme code used identifiers starting with `->`, which are a convenient choice for certain names. Therefore, R^{6}RS legalizes these identifiers. The separate production in the grammar is not particularly elegant. However, designing a more elegant production that does not overlap with representations of number objects or other lexeme classes has proven to be surprisingly difficult.

4.2. Comments

While R^{5}RS provides only the `;` syntax for comments, the report now describes three comment forms: In addition to `;`, `#|` and `|#` delimit block comments, and `#;` starts a "datum comment". (`#!r6rs` is also a kind of comment, albeit with a specific, fixed purpose.)

Block comments provide a convenient way of writing multiline comments, and are an often-requested and often-implemented syntactic addition to the language.

A datum comment always comments out a single datum—no more, and no less, something the other comment forms cannot reliably do. Their uses include commenting out alternative versions of a form and commenting out forms that may be required only in certain circumstances. Datum comments are perhaps most useful during development and debugging and may thus be less likely to appear in the final version of a distributed library or top-level program; even so, a programmer or group of programmers sometimes develop and debug a single piece of code concurrently on multiple systems, in which case a standard notation for commenting out a datum is useful.

4.3. Future extensions

The `#` is the prefix of several different kinds of syntactic entities: vectors, bytevectors, syntactic abbreviations related to syntax construction, nested comments, characters, `#!r6rs`, and implementation-specific extensions to the syntax that start with `#!`. In each case, the character following the `#` specifies what kind of syntactic datum follows. In the case of bytevectors, the syntax anticipates several different kinds of homogeneous vectors, even though R^{6}RS specifies only one. The `u8` after the `#v` identifies the components of the vector as unsigned 8-bit entities or octets.

5. Semantic concepts

5.1. Argument and subform checking

The report requires implementations to check the arguments of procedures and subforms for syntactic forms for adherence to the specification. However, implementations are not required to detect every violation of a specification. Specifically, the report allows the following exceptions:

1. Some restrictions are undecidable. Hence, checking is not required, such as certain properties of procedures passed as arguments, or properties of subexpressions, whose macro expansion may not terminate.

2. Checking that an argument is a list where doing so would be impractical or expensive is not required. Specifically, procedures that invoke another procedure passed as an argument are not required to check that a list remains a list after every invocation.

3. With some procedures, future extensions to the arguments they accept are explicitly allowed.

The second item deserves special attention, as the specific decisions made for the report are meant to enable "picky" implementations that catch as many violations and unportable assumptions made by programs as possible, while also enabling practical implementations that execute programs quickly.

5.2. Safety

R^{5}RS describes many situations not specified in the report as "is an error": Portable R^{5}RS programs cannot cause such situations, but R^{5}RS implementations are free to implement arbitrary behavior under this umbrella. Arbitrary behavior can include "crashing" the running program, or somehow compromising the integrity of its execution model to result in random behavior. This situation stands in sharp contrast to the common assumption that Scheme is a "safe" language, where each violation of a restriction of the language standard or the implementation would at least result in defined behavior (e.g., interrupting or aborting the program, or starting a debugger).

To avoid the problems associated with this arbitrary behavior, all libraries specified in the report must be safe, and they react to detected violations of the specification by raising an exception, which allows the program to detect and react to the violation itself.

The report allows implementations to provide "unsafe" libraries that may compromise safety.

5.3. Proper tail recursion

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

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

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

While a proper tail recursion has been a cornerstone property of Scheme since its inception, it is difficult to implement efficiently on some architectures, specifically those compiling to higher-level intermediate languages such as C or to certain virtual-machine architectures such as JVM or CIL.

Nevertheless, abandoning proper tail recursion as a language property and relegating it to optional optimizations would have far-reaching consequences: Many programs written with the assumption of proper tail recursion would no longer work. Moreover, the lack of proper tail recursion would prevent the natural expression of certain programming styles such as Actors-style message-passing systems, self-replacing servers, or automata written as mutually recursive procedures. Furthermore, if they did not exist, special "loop" constructs would have to be added to the language to compensate for the lack of a general iteration

construct. Consequently, proper tail recursion remains an essential aspect of the Scheme language.

6. Entry format

While it is reasonable to require the programmer to adhere to restrictions on arguments, some of these restrictions are either undecidable or too expensive to always enforce (see section 5.1). Therefore, some entries have an additional paragraph labelled "*Implementation responsibilities*" that distinguishes the responsibilities of the programmer from those of the implementation.

7. Libraries

The design of the library system was a challenging process: Many existing Scheme implementations offer "module systems", but they differ dramatically both in functionality and in the goals they address. The library system was designed with the primary requirement of allowing programmers to write, distribute, and evolve portable code. A secondary requirement was to be able to separately compile libraries in the sense that compiling a library requires only having compiled its dependencies. This entailed the following corollary requirements:

- Composing libraries requires management of dependencies.
- Libraries from different sources may have name conflicts. Consequently, name-space management is needed.
- Macro definitions appear in portable code, requiring that macro bindings may be exported from libraries, with all the consequences dictated by the referential-transparency property of hygienic macros.

The library system does not address the following goals, which were considered during the design process:

- independent compilation
- mutually dependent libraries
- separation of library interface from library implementation
- local modules and local imports

This section discusses some aspects of the design of the library system that have been controversial.

7.1. Syntax

A library definition is a single form, rather than a sequence of forms where some forms are some kind of header and the remaining forms contain the actual code. It is not clear that a sequence of forms is more convenient than a single form for processing and generation. Both syntactic choices have technical merits and drawbacks. The single-form syntax chosen for R^{6}RS has the advantage of being self-delimiting.

A difference between top-level programs and libraries is that a program contains only one top-level program but multiple libraries. Thus, delimiting the text for a library body will be a common need (in streams of various kinds) that it is worth standardizing the delimiters; parentheses are the obvious choice.

7.2. Local import

Some Scheme implementations feature module systems that allow a module's bindings to be imported into a local environment. While local imports can be used to limit the scope of an import and thus lead to more modular code and less need for the prefixing and renaming of imports, the existence of local imports would mean that the set of libraries upon which a library depends cannot be approximated as precisely from the library header. (The precise set of libraries used cannot be determined even in the absense of local import, because a library might be listed but its exports not used, and a library not listed might still be imported at run time through the `environment` procedure.) Leaving out local import for now does not preclude it from being added later.

7.3. Local modules

Some Scheme implementations feature local libraries and/or modules, e.g., libraries or modules that appear within top-level libraries or within local bodies. This feature allows libraries and top-level programs to be further subdivided into modular subcomponents, but it also complicates the scoping rules of the language. Whereas the library system allows bindings to be transported only from one library top level to another, local modules allow bindings to be transported from one local scope to another, which complicates the rules for determining where identifiers are bound. Leaving out local libraries and modules for now does not preclude them from being added later.

7.4. Fixed `import` and `export` clauses

The `import` and `export` clauses of the `library` form are a fixed part of the library syntax. This eliminates the need

to specify in what language or language version the clauses are written and simplifies the process of approximating the set of libraries upon which a library depends, as described in section 7.2. A downside is that `import` and `export` clauses cannot be abstracted, i.e., cannot be the products of macro calls.

7.5. Instantiation and initialization

Opinions vary on how libraries should be instantiated and initialized during the expansion and execution of library bodies, whether library instances should be distinguished across phases, and whether levels should be declared so that they constrain identifier uses to particular phases. This report therefore leaves considerable latitude to implementations, while attempting to provide enough guarantees to make portable libraries feasible.

Note that, if each right-hand side of the keyword definition and keyword binding forms appearing in a program is a `syntax-rules` or `identifier-syntax` form, `syntax-rules` and `identifier-syntax` forms do not appear in any other contexts, and no `import` form employs `for` to override the default import phases, then the program does not depend on whether instances are distinguished across phases, and the phase of an identifier's use cannot be inconsistent with the identifier's level. Moreover, the phase of an identifier's use is never inconsistent with the identifier's level if the implementation uses an implicit phasing model in which references are allowed at any phase regardless of any phase declarations.

7.6. Immutable exports

The asymmetry in the prohibitions against assignments to explicitly and implicitly exported variables reflects the fact that the violation can be determined for implicitly exported variables only when the importing library is expanded.

7.7. Compound library names

Library names are compound. This differs from the treatment of identifiers in the rest of the language. Using compound names reflects experience across programming languages that a structured top-level name space is necessary to avoid collisions. Embedding a hierarchy within a single string or symbol is certainly possible. However, in Scheme, list data is the natural means for representing hierarchical structure, rather than encoding it in a string or symbol. The hierarchical structure makes it easy to formulate policies for choosing unique names or possible storage formats in a file system. See appendix G. Consequently, despite the syntactic complexity of compound names, and despite the potential mishandling of the hierarchy by implementations, the editors chose the list representation.

7.8. Versioning

Libraries and `import` clauses optionally carry versioning information. This allows reflecting the development history of a library, but also significantly increases the complexity of the library system. Experience with module systems gathered in other languages as well as with shared libraries at the operating-system level consistently indicates that relying only on the name of a module for identification causes conflicts impossible to rectify in the absence of versioning information, and thus diminishes the opportunities for sharing code. Therefore, versioning is part of the library system.

7.9. Treatment of different versions

Implementations are encouraged to prohibit two libraries with the same name but different versions to coexist within the same program. While this prevents the combination of libraries and programs that require different versions of the same library, it eliminates the potential for having multiple copies of a library's state, thus avoiding problems experienced with other shared-library mechanisms, including Windows DLLs and Unix shared objects.

8. Top-level programs

The notion of "top-level program" is new in R^{6}RS and replaces the notion of "Scheme program" in R^{5}RS. The two are quite different: While a R^{6}RS top-level program is defined to be a complete, textual entity, an R^{5}RS program can evolve by being entered piecemeal into a running Scheme system. Many Scheme systems have interactive command-line environments based on the semantics of R^{5}RS programs. However, the specification of R^{5}RS programs is not really sufficient to describe how to operate an arbitrary Scheme system: The R^{5}RS is ambiguous on some aspects of the semantics such as binding. Moreover, R^{5}RS's `load` procedure does say how to load source code into the running system; the pragmatics of `load` would often make compiling programs before execution problematic, in particular with regard to macros. Furthermore, Scheme implementations handle treatment of and recovery from errors in different ways.

Tightening the specification of programs from R^{5}RS would have been possible, but could have restricted the design employed by Scheme implementations in undesirable ways. Moreover, alternative approaches to structuring the user interface of a Scheme implementation have emerged since R^{5}RS. Consequently, R^{6}RS makes no attempt at trying to

specify the semantics of programs as in R[5]RS; the design of an interactive environment is now completely in the hands of the implementors. On the other hand, being able to distribute portable programs is one of the goals of the R[6]RS process. As a result, the notion of top-level program was added to the report.

By allowing the interleaving of definitions and expressions, top-level programs support exploratory and interactive development, without imposing unnecessary organizational overhead on code that might not be intended for reuse.

9. Primitive syntax

9.1. Unspecified evaluation order

The order in which the subexpressions of an application are evaluated is unspecified, as is the order in which certain subexpressions of some other forms such as `letrec` are evaluated. While this causes occasional confusion, it encourages programmers to write programs that do not depend on a specific evaluation order, and thus may be easier to read. Moreover, it allows the programmer to express that the evaluation order really does not matter for the result. A secondary consideration is that some compilers are able to generate better code if they can choose evaluation order.

10. Expansion process

The description of macro expansion in R[6]RS is considerably more involved than in R[5]RS: One reason is that the specification of expansion in R[5]RS is ambiguous in several important respects. For example, R[5]RS does not specify whether `define` is a binding form. Also, it was not clear whether definitions of macros had to precede their uses. The fact that the set of available bindings may influence the matching process of macro expansion further complicates matters. The specific algorithm R[6]RS describes is one of the simplest expansion strategies that addresses these questions. It has the advantage that it visits every subform of the source code only once.

The description of the expansion process specifically avoids specifying the recursive case, where a macro use expands into a definition whose binding would influence the expansion of the macro use after the fact, as this might lead to confusing programs. Implementations should detect such cases as syntax violations.

11. Base library

11.1. Library organization

The libraries of the Scheme standard are organized according to projected use. Hence, the `(rnrs base (6))` library exports procedures and syntactic abstractions that are likely to be useful for most Scheme programs and libraries. Conversely, each of the libraries relegated to the separate report on libraries is likely to be missing from the imports of a substantial number of programs and libraries. Naturally, the specific decisions about this organization and the separation of concerns of the libraries are debatable, and represent a best attempt of the editors.

A number of secondary criteria were also used in choosing the exports of the base library. In particular, macros transformers defined using the facilities of the base library are guaranteed to be hygienic; hygiene-breaking transformers are only available through the `(rnrs syntax-case (6))` library.

Note that `(rnrs base (6))` is not a "primitive library" in the sense that all other libraries of the Scheme standard can be implemented portably using only its exports. Moreover, the library organization is generally not layered from more primitive to more advanced libraries, even though some libraries can certainly be implemented in terms of others. Such an organization would have little benefit for users and may not reflect the internal organization of any particular implementation. Instead, libraries are organized by use.

The distinction between primitive and derived features was removed from the report for similar reasons.

11.2. Bodies

In library bodies and local bodies, all definitions must precede all expressions. R[6]RS treats bodies in top-level programs as a special case. Allowing definitions and expressions to be mixed in top-level programs has ugly semantics, and introduces a special case, but was allowed as a concession to convenience when constructing programs rapidly via cut and paste.

Definitions are not interchangeable with expressions, so definitions cannot be allowed to appear wherever expressions can appear. Composition of definitions with expressions therefore must be restricted in some way. The question is what those restrictions should be.

Historically, top-level definitions in Scheme have had a different semantics from definitions in bodies. In a body, definitions serve as syntactic sugar for the bindings of a `letrec` (or `letrec*` in R[6]RS) that is implicit at the head of every body.

That semantics can be stretched to cover top-level programs by converting expressions to definitions of ignored variables, but does not easily generalize to allow definitions to be placed anywhere within expressions. Different generalizations of definition placement are possible, however a survey of current Scheme code found surprisingly few places where such a generalization would be useful.

If such a generalization were adopted, programmers who are familiar with Java and similar languages might expect definitions to be allowed in the same kinds of contexts that allow declarations in Java. However, Scheme definitions have `letrec*` scope, while Java declarations (inside a method body) have `let*` scope and cannot be used to define recursive procedures. Moreover, Scheme's `begin` expressions do not introduce a new scope, while Java's curly braces do introduce a new scope. Also, flow analysis is nontrivial in higher order languages, while Java can use a trivial flow analysis to reject programs with undefined variables. Furthermore, Scheme's macro expander must locate all definitions, while Java has no macro system. And so on. Rather than explain how those facts justify restricting definitions to appear as top-level forms of a body, it is simpler to explain that definitions are just syntactic sugar for the bindings of an implicit `letrec*` at the head of each body, and to explain that the relaxation of that restriction for top-level bodies is (like several other features of top-level programs) an ad-hoc special case.

11.3. Export levels

The `syntax-rules` and `identifier-syntax` forms are used to create macro transformers and are thus needed only at expansion time, i.e., meta level 1.

The identifiers `unquote`, `unquote-splicing`, `=>`, and `else` serve as literals in the syntax of one or more syntactic forms; e.g., `else` serves as a literal in the syntax of `cond` and `case`. Bindings of these identifiers are exported from the base library so that they can be distinguished from other bindings of these identifiers or renamed on import. The identifiers `...`, `_`, and `set!` serve as literals in the syntax of `syntax-rules` and `identifier-syntax` forms and are thus exported along with those forms with level 1.

11.4. Binding forms

The `let-values` and `let-values*` forms are compatible with SRFI 11 [16].

11.5. Equivalence predicates

11.5.1. Treatment of procedures

The definition of `eqv?` allows implementations latitude in their treatment of procedures: implementations are free either to detect or to fail to detect that two procedures are equivalent to each other, and can decide whether or not to merge representations of equivalent procedures by using the same pointer or bit pattern to represent both. Moreover, they can use implementation techniques such as inlining and beta reduction that duplicate otherwise equivalent procedures.

11.5.2. Equivalence of NaNs

The basic reason why the behavior of `eqv?` is not specified on NaNs is that the IEEE-754 standard does not say much about how the bits of a NaN are to be interpreted, and explicitly allows implementations of that standard to use most of a NaN's bits to encode implementation-dependent semantics. The implementors of a Scheme system should therefore decide how `eqv?` should interpret those bits.

Arguably, R^{6}RS should require

```
(let ((x ⟨expression⟩)) (eqv? x x))
```

to evaluate to `#t` when ⟨expression⟩ evaluates to a number object; both R^{5}RS and R^{6}RS imply this for certain other types, and for most numbers objects, but not for NaNs. Since the IEEE 754 and draft IEEE 754R [19] both say that the interpretation of a NaN's payload is left up to implementations, and implementations of Scheme often do not have much control over the implementation of IEEE arithmetic, it would be unwise for R^{6}RS to insist upon the truth of

```
(let ((x ⟨expression⟩))
  (or (not (number? x))
      (eqv? x x)))
```

even though that expression is likely to evaluate to `#t` in most systems. For example, a system with delayed boxing of inexact real number objects might box the two arguments to `eqv?` separately, the boxing process might involve a change of precision, and the two separate changes of precision may result in two different payloads.

When x and y are flonums represented in IEEE floating point or similar, it is reasonable to implement `(eqv?` x y`)` by a bitwise comparison of the floating-point representations. R^{6}RS should not require this, however, because

1. R^{6}RS does not require that flonums be represented by a floating-point representation,
2. the interpretation of a NaN's payload is explicitly implementation-dependent according to both the IEEE-754 standard and the current draft of its proposed replacement, IEEE 754R, and
3. the semantics of Scheme should remain independent of bit-level representations.

For example, IEEE 754, IEEE 754R, and the draft R^{6}RS all allow the external representation `+nan.0` to be read as a NaN whose payload encodes the input port and position at which `+nan.0` was read. This is no different from any other external representation such as `()`, `#()`, or `324`. An implementation can have arbitrarily many bit-level representations of the empty vector, for example, and some do.

That is why the behavior of the `eq?` and `eqv?` procedures on vectors cannot be defined by reference to bit-level representations, and must instead be defined explicitly.

11.5.3. `eq?`

It is usually possible to implement `eq?` much more efficiently than `eqv?`, for example, as a simple pointer comparison instead of as some more complicated operation. One reason is that it may not be possible to compute `eqv?` of two number objects in constant time, whereas `eq?` implemented as pointer comparison will always finish in constant time. The `eq?` predicate may be used like `eqv?` in applications using procedures to implement objects with state since it obeys the same constraints as `eqv?`.

11.6. Arithmetic

11.6.1. Full numerical tower

R^{5}RS does not require implementations to support the full numeric tower. Consequently, writing portable R^{5}RS programs that perform substantial arithmetic is difficult; it is unnecessarily difficult even to write programs whose arithmetic is portable between different implementations in the same category. The portability problems were most easily solved by requiring all implementations to support the full numerical tower.

11.6.2. IEEE-754 conformance

As mentioned in chapter 3, the treatment of infinities, NaNs and -0.0, if present in a Scheme implementation, are in line with IEEE 754 [18] and IEEE 754R [19]. Analogously, the specification of branch cuts for certain transcendental functions have been changed from R^{5}RS to conform to the IEEE standard.

11.6.3. Transcendental functions

The specification of the transcendental functions follows Steele [37], which in turn cites Penfield [26]; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions.

11.6.4. Domains of numerical predicates

The domains of the `finite?`, `infinite?`, and `nan?` procedures could be expanded to include all number objects, or perhaps even all objects. However, R^{6}RS restricts them to real number objects. Expanding `nan?` to complex number objects would involve at least some arbitrariness; not expanding its domain while expanding the domains of the other two would introduce an irregularity into the domains of these three procedures, which are likely to be used together. It is easier for programmers who wish to use these procedures with complex number objects to express their intent in terms of the real-only versions than it would be for the editors to guess their intent.

11.6.5. Numerical types

Scheme's numerical types are the exactness types exact and inexact, the tower types integer, rational, real, complex, and number, and the Cartesian product of the exactness types with the tower types, where $\langle t_1, t_2 \rangle$; is regarded as a subtype of both t_1 and t_2.

These types have an aesthetic symmetry to them, but they are not equally important In practice, there is reason to believe that the most important numerical types are the exact integer objects, the exact rational number objects, the inexact real number objects, and the inexact complex number objects. This section explores one of the reasons those four types are important in practice, and why real number objects have an exact zero as their imaginary part in R^{6}RS (a change from R^{5}RS).

11.6.6. Closure Properties

Each of the four types mentioned above corresponds to a set of values that turns up repeatedly as the natural domain or range of the functions that are computed by Scheme's standard procedures. The reason these types turn up so often is that they are closed under certain sets of operations.

The exact integer objects, for example, are closed under the integral operations of addition, subtraction, and multiplication. The exact rational number objects are closed under the rational operations, which consist of the integral operations plus division (although division by zero is a special case). The real number objects (and inexact real number objects) are closed under some (often inexact) interpretation of rational and irrational operations such as `exp` and `sin`, but are not closed under operations such as `log`, `sqrt`, and `expt`. The complex (and inexact complex) number objects are closed under the largest set of operations.

Representation-specific operations

A naive implementation of Scheme's arithmetic operations is slow compared to the arithmetic operations of most other languages, mainly because most operations must perform case dispatch on the representation types of their arguments. The potential for this case dispatch arises when the type of an operation's argument is represented by a

union of two or more representation types, or because the operation must raise an exception when given an argument of an incorrect type. (The second reason can be regarded as a special case of the first.)

To make Scheme's arithmetic more efficient, many implementations provide sets of operations whose domain is restricted to a single representation type, and which are not expected to raise an exception when given arguments of incorrect type when used in an unsafe mode.

Alternatively, or in addition, several compilers perform a flow analysis that attempts to infer the representation types of expressions. When a single representation type can be inferred for each argument of an operation, and those types match the types expected by some representation-specific version of the operation, then the compiler can substitute the specific version for the more general version that was specified in the source code.

Flow analysis

Flow analysis is performed by solving the type and interval constraints that arise from such things as:

- the types of literal constants, e.g. `2` is an exact integer object that is known to be within the interval $[2, 2]$
- conditional control flow that is predicated on known inequalities, e.g., `(if (< i n)` ⟨expression$_1$⟩ ⟨expression$_2$⟩`)`
- conditional control flow that is predicated on known type predicates, e.g., `(if (real? x)` ⟨expression$_1$⟩ ⟨expression$_2$⟩`)`
- the closure properties of known operations (for example, `(+` *flonum*$_1$ *flonum*$_2$`)` always evaluates to a flonum)

The purpose of flow analysis (as motivated in this section) is to infer a single representation type for each argument of an operation. That places a premium on predicates and closure properties from which a single representation type can be inferred.

In practice, the most important single representation types are fixnum, flonum, and compnum. (A compnum is a pair of flonums, representing an inexact complex number object.) These are the representation types for which a short sequence of machine code can be generated when the representation type is known, but for which considerably less efficient code will probably have to be generated when the representation type cannot be inferred.

The fixnum representation type is not closed under any operation of R^{5}RS, so it is hard for flow analysis to infer the fixnum type from portable code. Sometimes the combination of a more general type (e.g., exact integer object) and an interval (e.g., $[0, n)$, where n is known to be a fixnum) can imply the fixnum representation type. Adding fixnum-specific operations that map fixnums to fixnums greatly increases the number of fixnum representation types that a compiler can infer.

The flonum representation type is not closed under operations such as `sqrt` and `expt`, so flow analysis tends to break down in the presence of those operations. This is unfortunate, because those operations are normally used only with arguments for which the result is expected to be a flonum. Adding flonum-specific versions such as `flsqrt` and `flexpt` improves the effectiveness of flow analysis.

R^{5}RS creates a more insidious problem by defining `(real?` *z*`)` to be true if and only if `(zero? (imag-part` *z*`))` is true. This means, for example, that `-2.5+0.0i` is real. If `-2.5+0.0i` is represented as a compnum, then the compiler cannot rely on `x` being a flonum in the consequent of `(if (real? x)` ⟨expression$_1$⟩ ⟨expression$_2$⟩`)`. This problem could be fixed by writing all of the arithmetic operations so that any compnum with a zero imaginary part is converted to a flonum before it is returned, but that merely creates an analogous problem for compnum arithmetic, as explained below. R^{6}RS adopted a proposal by Brad Lucier to fix the problem: `(real?` *z*`)` is now true if and only if `(imag-part` *z*`)` is an exact zero.

The compnum representation type is closed under virtually all operations, provided no operation that accepts two compnums as its argument ever returns a flonum. To work around the problem described in the paragraph above, several implementations automatically convert compnums with a zero imaginary part to the flonum representation. This practice virtually destroys the effectiveness of flow analysis for inferring the compnum representation, so it is not a good workaround. To improve the effectiveness of flow analysis, it is better to change the definition of Scheme's real number objects as described in the paragraph above.

div and mod

Given arithmetic on exact integer objects of arbitrary precision, it is a trivial matter to derive signed and unsigned integer types of finite range from it by modular reduction. For example 32-bit signed two-complement arithmetic behaves like computing with the residue classes "mod 2^{32}", where the set $\{-2^{31}, \ldots, 2^{31} - 1\}$ has been chosen to represent the residue classes. Likewise, unsigned 32-bit arithmetic also behaves like computing "mod 2^{32}", but with a different set of representatives $\{0, \ldots, 2^{32} - 1\}$.

Unfortunately, the R^{5}RS operations `quotient`, `remainder`, and `modulo` are not ideal for this purpose. In the following example, `remainder` fails to transport the additive

group structure of the integers over to the residues modulo 3.

```
(remainder (+ -2 3) 3)      ⟹ 1,
(remainder (+ (remainder -2 3)
              (remainder 3 3))
           3)               ⟹ -2
```

In fact, `modulo` should have been used, producing residues in $\{0, 1, 2\}$. For modular reduction with symmetric residues, i.e., in $\{-1, 0, 1\}$ in the example, it is necessary to define a more complicated reduction altogether.

Therefore, `quotient`, `remainder`, and `modulo` have been replaced in R[6]RS by the `div`, `mod`, `div0`, and `mod0` procedures, which are more useful when implementing modular reduction. The underlying mathematical functions div, mod, div_0, and mod_0 (see report section 11.7.3) have been adapted from the div and mod operations by Egner et al. [11]. They differ in the representatives from the residue classes they return: div and mod always compute a nonnegative residue, whereas div_0 and mod_0 compute a residue from a set centered on 0. The former can be used, for example, to implement unsigned fixed-width arithmetic, whereas the latter correspond to two's-complement arithmetic.

These operations differ slightly from the div and mod operations from Egner et al. The latter make both operations available through a single pair of operations that distinguish between the two cases for residues by the sign of the divisor (as well as returning 0 for a zero divisor). Splitting the operations into two sets of procedures avoids potential confusion.

The procedures `modulo`, `remainder`, and `quotient` from R[5]RS can easily be defined in terms of `div` and `mod`.

11.6.7. Numerical predicates

The behavior of the numerical type predicates `complex?`, `real?`, `rational?`, and `integer?` is motivated by closure properties described in section 11.6.6. Conversely, the procedures `real-valued?`, `rational-valued?`, and `integer-valued?` test whether a given number object can be coerced to the specified type without loss of numerical accuracy.

11.6.8. Notes on individual procedures

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

`sqrt` The behavior of `sqrt` is consistent with the IEEE floating-point standard.

`number->string` If z is an inexact number object represented using binary floating point, and the radix is 10, then the expression listed in the specification is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and representations other than binary floating-point.

11.7. Characters and strings

While R[5]RS specifies characters and strings in terms of its own, limited character set, R[6]RS specifies characters and strings in terms of Unicode. The primary goal of the design change was to improve the portability of Scheme programs that manipulate text, while preserving a maximum of backward compatibility with R[5]RS.

R[6]RS defines characters to be representations of Unicode scalar values, and strings to be indexed sequences of characters. This is a different representation for Unicode text than the representations chosen by some other programming languages such as Java or C#, which use UTF-16 code units as the basis for the type of characters.

The representation of Unicode text corresponds to the lowest semantic level of the Unicode standard: The Unicode standard specifies most semantic properties in terms of Unicode scalar values. Thus, Unicode strings in Scheme allow the straightforward implementation of semantically sensitive algorithms on strings in terms of these scalar values.

In contrast, UTF-16 is a specific encoding for Unicode text, and performing semantic manipulation on UTF-16 representations of text is awkward. Choosing UTF-16 as the basis for the string representation would have meant that a character object potentially carries no semantic information at all, as surrogates have to be combined pairwise to yield the corresponding Unicode scalar value. (As a result, Java provides some semantic operations on Unicode text in two overloadings, one for character objects and one for integers that are Unicode scalar values.)

The surrogates cover a numerical range deliberately omitted from the set of Unicode scalar values. Hence, surrogates have no representation as characters—they are merely an artifact of the design of UTF-16. Including surrogates in the set of characters introduces complications similar to the complications of using UTF-16 directly. In particular, most Unicode consortium standards and recommendations explicitly prohibit unpaired surrogates, including the UTF-8 encoding, the UTF-16 encoding, the UTF-32 encoding, and recommendations for implementing the ANSI C `wchar_t` type. Even UCS-4, which originally permitted a larger range of values that includes the surrogate range, has been redefined to match UTF-32 exactly. That is, the original UCS-4 range was shrunk and surrogates were excluded.

Arguably, a higher-level model for text could be used as the basis for Scheme's character and string types, such as

grapheme clusters. However, no design satisfying the goals stated above was available when the report was written.

11.8. Symbols

Symbols have exactly the properties needed to represent identifiers in programs, and so most implementations of Scheme use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated values are used in C and Pascal.

11.9. Control features

11.9.1. `call-with-current-continuation`

A common use of `call-with-current-continuation` is for structured, non-local exits from loops or procedure bodies, but in fact `call-with-current-continuation` is useful for implementing a wide variety of advanced control structures.

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [23] invented a general-purpose escape operator called the J-operator. John Reynolds [28] described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the `catch` construct could be provided by a procedure instead of by a special syntactic construct, and the name `call-with-current-continuation` was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name `call/cc` instead.

11.9.2. `dynamic-wind`

The `dynamic-wind` procedure was added more recently in R^{5}RS. It enables implementing a number of abstractions related to continuations, such as implementing a general dynamic environment, and making sure that finalization code runs when some dynamic extent expires. More generally, the `dynamic-wind` procedure provides a guarantee that

`(dynamic-wind` *before thunk after*`)`

cannot call *thunk* unless *before* has been called, and it cannot leave the dynamic extent of the call to *thunk* without calling *after*. These evaluations are never nested. As this guarantee is crucial for enabling many of the uses of `call-with-current-continuation` and `dynamic-wind`, both are specified jointly.

11.9.3. Multiple values

Many computations conceptually return several results. Scheme expressions implementing such computations can return the results as several values using the `values` procedure. Of course, such expressions could alternatively return the results as a single compound value, such as a list, vector, or a record. However, values in programs usually represent conceptual wholes; in many cases, multiple results yielded by a computation lack this coherence. Moreover, this would be inefficient in many implementations, and a compiler would need to perform significant optimization to remove the boxing and unboxing inherent in packaging multiple results into a single values. Most importantly, the mechanism for multiple values in Scheme establishes a standard policy for returning several results from an expression, which makes constructing interfaces and using them easier.

R^{6}RS does not specify the semantics of multiple values completely. In particular, it does not specify what happens when several (or zero) values are returned to a continuation that implicitly accepts only one value. In particular:

```
((lambda (x) x) (values 1 2))
```
$\Longrightarrow$ *unspecified*

Whether an implementation must raise an exception when evaluating such an expression, or should exhibit some other, non-exceptional behavior is a contentious issue. Variations of two different and fundamentally incompatible positions on this issue exist, each with its own merits:

1. Passing the wrong number of values to a continuation is typically a violation, one that implementations ideally detect and report.
2. There is no such thing as returning the wrong number of values to a continuation. In particular, continuations not created by `begin` or `call-with-values` should ignore all but the first value, and treat zero values as one unspecified value.

R^{6}RS allows an implementation to take either position. Moreover, it allows an implementation to let `set!`, `vector-set!`, and other effect-only operators to pass zero values to their continuations, preventing a program from making obscure use of the return value. This causes a potential compatibility problem with R^{5}RS, which specifies that such expression return a single unspecified value, but the benefits of the change were deemed to outweigh the costs.

11.10. Macro transformers

11.10.1. `syntax-rules`

While the first subform of ⟨srpattern⟩ of a ⟨syntax rule⟩ in a `syntax-rules` form (see report section 11.19) may be

an identifier, the identifier is not involved in the matching and is not considered a pattern variable or literal identifier. This is actually important, as the identifier is most often the keyword used to identify the macro. The scope of the keyword is determined by the binding form or syntax definition that binds it to the associated macro transformer. If the keyword were a pattern variable or literal identifier, then the template that follows the pattern would be within its scope regardless of whether the keyword were bound by `let-syntax`, `letrec-syntax`, or `define-syntax`.

12. Formal semantics

The operational semantics in report chapter A replaces the denotational semantics in R^{5}RS. The denotational semantics in R^{5}RS has several problems, most seriously its incomplete treatment of the unspecific evaluation order of applications: the denotational semantics suggests that a single unspecified order is used. Modelling nondeterminism is generally difficult with denotational semantics, and an operational semantics allows specifying the unspecified evaluation order precisely.

13. Unicode

13.1. Case mapping

The various case-mapping procedures of the `(rnrs unicode (6))` library all operate in a locale-independent manner. The Unicode standard also offers locale-sensitive case operations, not implemented by the procedures from the `(rnrs unicode (6))` library. While the library does not make available the full spectrum of case-related functionality defined by the Unicode standard, it does provide the most commonly used procedures. In particular, this strategy has allowed providing procedures mostly compatible with those provided by R^{5}RS. (A minor exception is the case-insensitive procedures for string comparison. However, it is unlikely that this affects many existing programs.) Providing locale-sensitive operations would have meant significant novel design effort without significant precedent, which is why they are not part of R^{6}RS.

The case-mapping procedures operating on characters are not sufficient for implementing case mapping on strings. For example, the upper-case version of the German "ß" in a string is "SS". As `char-upcase` can only return a single character, it must return ß for ß. This limits the usefulness of the procedures operating on characters, but provides compatibility with R^{5}RS sufficient for many existing applications. Moreover, it provides direct access to the corresponding attributes of the Unicode character database.

14. Bytevectors

Bytevectors are a representation for binary data, based on SRFI 74 [32]. The primary motivation for including them in R^{6}RS was to enable binary I/O. Positions in bytevectors always refer to certain bytes or octets. However, the operations of the `(rnrs bytevectors (6))` library provide access to binary data in various byte-aligned formats, such as signed and unsigned integers of various widths, IEEE floating-point representations, and textual encodings. This differs notably from representations for binary data as homogeneous vectors of numbers. In settings related to I/O, an application often needs to access different kinds of entities from a single binary block. Providing operations for them on a single datatype considerably reduces both programming effort and library size.

Bytevectors can also be used to encode sequences of unboxed number objects. Unencapsulated use of bytevectors for this purpose may lead to aliasing, which may reduce the effectiveness of compiler optimizations. However, sealedness and opacity of records, together with bytevectors, make it possible to construct a portable implementation for new data types that provides fast and memory-efficient arrays of homogeneous numerical data.

15. List utilities

The `(rnrs lists (6))` library provides a small number of useful procedures operating on lists, including several procedures from R^{5}RS. The goal of the library is to provide only procedures likely to be useful for many programs. Consequently, the selection represented by `(rnrs lists (6))` is less exhaustive than the widely implemented SRFI 1 [30]. Several changes were made with respect to the corresponding procedures SRFI 1 to simplify the specification, and to establish uniform naming conventions.

15.1. Notes on individual procedures

`memp`, `member`, `memv`, and `memq` Although they are ordinarily used as predicates, `memp`, `member`, `memv`, and `memq`, do not have question marks in their names, because they return useful values rather than just `#t` or `#f`.

16. Sorting

The procedures of the `(rnrs sorting (6))` library provide simple interfaces to sorting algorithms useful to many programs. In particular, `list-sort` and `vector-sort` guarantee stable sorting using $O(n \lg n)$ calls to the comparison procedure. Straightforward implementations of merge sort [7] have the desired properties. Note that, at least with merge sort, stability carries no significant implementation or performance burden.

The choice of "strictly less than" for the comparison relation is consistent with the most common choice of existing

Scheme libraries for sorting. Moreover, using a procedure returning three possible values (for less than, equal, and greater than) instead of a boolean comparison procedure would make calling the sorting procedures less convenient, with no discernible performance advantage.

The specification of the `vector-sort!` procedure is meant to allow an implementation using quicksort [17], hence the $O(n^2)$ bound on the number of calls to the comparison procedure, and the omission of the stability requirement.

17. Control structures

17.1. `when` and `unless`

The `when` and `unless` forms are syntactic sugar for one-armed `if` expressions. Because each incorporates an implicit `begin`, they are sometimes more convenient than one-armed `if`. Some programmers always use `when` and `unless` in lieu of one-armed `if` to make clear when a one-armed conditional is being used.

17.2. `case-lambda`

The `case-lambda` form allows constructing procedures that distinguish different numbers of arguments. Using `case-lambda` makes this considerably easier than deconstructing a list containing optional arguments explicitly. Moreover, Scheme implementations might optimize dispatch on the number of arguments when expressed as `case-lambda`, which is considerably harder for code that explicitly deconstructs argument lists.

18. Records

18.1. Syntactic layer

While the syntactic layer can be expressed portably in terms of the procedural layer, standardizing a particular surface syntax facilitates communication via code.

Moreover, the syntactic layer is designed to allow expansion-time determination of record characteristics, including field offsets, so that, for example, record accesses can be reduced to simple memory indirects without flow analyses or any other nontrivial compiler support. (This property may be lost if the `parent-rtd` clause is present, and the parent is thus not generally known until run time.) Thus, the syntactic layer facilitates the development of efficient portable libraries that define and use record types and can serve as a basis for other syntactic record definition constructs.

18.2. Positional access and field names

The record and field names passed to `make-record-type-descriptor` and appearing in the syntactic layer are for informational purposes only, e.g., for printers and debuggers. In particular, the accessor and mutator creation routines do not use names, but rather field indices, to identify fields. Thus, field names are not required to be distinct in the procedural or syntactic layers. This relieves macros and other code generators from the need to generate distinct names.

Moreover, not requiring distinctness prevents naming conflicts that occur when a field in a base type is renamed such that it is the same as in an extension Also, the record and field names are used in the syntactic layer for the generation of accessor and mutator names, and thus duplicate field names may lead to accessor and mutator naming conflicts.

18.3. Lack of multiple inheritance

Multiple inheritance was considered but omitted from the records facility, as it raises a number of semantic issues such as sharing among common parent types.

18.4. Constructor mechanism

The constructor-descriptor mechanism is an infrastructure for creating specialized constructors, rather than just creating default constructors that accept the initial values of all the fields as arguments. This infrastructure achieves full generality while leaving each level of an inheritance hierarchy in control over its own fields and allowing child record definitions to be abstracted away from the actual number and contents of parent fields.

The constructor mechanism allows the initial values of the fields to be specially computed or to default to constant values. It also allows for operations to be performed on or with the resulting record, such as the registration of a record for finalization. Moreover, the constructor-descriptor mechanism allows the creation of such initializers in a modular manner, separating the initialization concerns of the parent types from those of the extensions.

18.5. Sealed record types

Record types may be sealed. This feature allows enforcing abstraction barriers, which is useful in itself, but also allows more efficient compilation.

In particular, when the implementor of an abstract data type chooses to represent that ADT by a record type, and allows one of the record types that represent the ADT to be

exposed and extended, then the ADT is no longer abstract. Its implementors must expose enough information to allow for effective subtyping, and must commit to enough of the representation to allow those subtypes to continue to work even as the ADT evolves.

A partial solution is to maintain independence of the child record type from the specific fields of the parent, particularly by specifying a record constructor descriptor for the parent type that is independent of its specific fields. When this is deemed to be insufficient, the record type can be sealed, thereby preventing the ADT from being subtyped. (This does not completely eliminate the problem, however, since the ADT may be extended implicitly, i.e., used as a delegate for some other type.)

Moreover, making a record type sealed may prevent its accessors and mutators from becoming polymorphic, which would make effective flow analysis and optimization difficult. This is particularly relevant for Scheme implementations that use records to implement some of Scheme's other primitive data types such as pairs.

19. Conditions and exceptions

19.1. Exceptions

The goals of the exception mechanism are to help programmers share code which relies on exception handling, and to provide information on violations of specifications of procedures and syntactic forms. This exception mechanism is a extension of SRFI 34 [22][1], which was primarily designed to meet the first goal. However, it has proven suitable for addressing the second goal of dealing with violations as well. (More on the second goal below in the discussion of the condition system.)

For some violations such the use of unsupported NaNs or infinities, as well as other applications, an exception handler may be able to repair the cause of the exception, for example by substituting a suitable object for the NaN or infinity. Therefore, the exception mechanism extends SRFI 34 by continuable exceptions, and specifies the continuation of an exception handler

19.2. Conditions

Conditions are values that communicate information about exceptional situations between parts of a program. Code that detects an exception may be in a different part of the program than the code that handles it. In fact, the former may have been written independently from the latter. Consequently, to facilitate effective handling of exceptions, conditions should communicate as much information as possible as accurately as possible, and still allow effective handling by code that did not precisely anticipate the nature of the exception that has occurred.

The `(rnrs conditions (6))` library provides two mechanisms to enable this kind of communication:

- subtyping (through record types) among condition types allows handling code to determine the general nature of an exception even though it does not anticipate its exact nature,
- compound conditions allow an exceptional situation to be described in multiple ways.

As an example, a networking error that occurs during a file operation on a remote drive fits two descriptions: "networking error" and "file-system error". An exception handler might only look for one of the two. Compound conditions are a simple solution to this problem. Moreover, compound conditions also make providing auxiliary information as part of the condition object, such as an error message, easier.

The standard condition hierarchy makes an important distinction between *errors* and *violations*: An error is an exceptional situation in the environment, which the program cannot avoid or prevent. For example, I/O errors are represented by condition types that are subtypes of `&error`. Violations, on the other hand, are exceptional situations that the program could have avoided. Violations are typically programming mistakes. The distinction between the two is not always clear, and it may be possible but inordinately difficult or expensive to detect certain violations. The use of `eval` also blurs the distinction. Nevertheless, many cases do allow distinguishing between errors and violations. Consequently, exception handlers that handle errors are common, whereas programmers should introduce exception handlers that handle violations with great care.

20. I/O

20.1. File names

The file names in most common operating systems, despite their appearance in most cases, are not text: For example, Unix uses null-terminated byte sequences, and Windows uses null-terminated sequences of UTF-16 code units. On Unix, the textual representation of a file name depends on the locale, an environmental setting. In both cases, a file name may be an invalid encoding and thus not correspond to a string. An appropriate representation for file names that covers these cases while still offering convenient access to file-system names through strings is still an open problem. Therefore, R[6]RS allows specifying file names as strings, but also allows an implementation to add its own representation for file names.

[1] There is also a small different to SRFI 34, namely that `guard` calls `raise-continuable` instead of `raise` when re-raising an exception.

20.2. File options

The flags specified for `file-options` represent only a common subset of meaningful options on popular platforms. The `file-options` form does not restrict the ⟨file-options name⟩s, so implementations can extend the file options by platform-specific flags.

20.3. End-of-line styles

The set of end-of-line styles recognized by the `(rnrs io ports (6))` library is not closed, because end-of-line styles other than those listed might become commonplace in the future.

20.4. Error-handling modes

The set of error-handling modes is not closed, because implementations may support error-handling modes other than those listed.

20.5. Binary and textual ports

The plethora of widely used encodings for texts makes providing textual I/O significantly more complicated than the simple model offered by R^{5}RS. In particular, realistic textual I/O should address encodings such as UTF-16 that include a header word determining the "actual" encoding of the rest of the byte stream, stateful encodings, and textual formats such as XML, which specify the encoding in a header line. Consequently, a library implementing textual I/O should support specifying an encoding upon opening a port, but should also support opening a port in "binary mode" to determine the encoding and switch to "text mode".

In contrast, arbitrary switching between "binary mode" and "text mode" is difficult to support, as it may interfere with efficient buffering strategies, and because the semantics may be unclear in the case of stateful encodings. Consequently, the `(rnrs io ports (6))` library allows switching from "binary mode" to "text mode" by converting a binary port into a textual port, but not the other way around. The `transcoded-port` procedure closes the binary port to preclude interference between the binary port and the textual port constructed from it. Applications that read from sources that intersperse binary and textual data should open a binary port and use either `bytevector->string` or the procedures from the `(rnrs bytevectors (6))` library to convert the binary data to text.

The separation of binary and textual ports enables creating ports from both binary and textual sources and sinks. It also makes creating both binary and textual versions of many procedures necessary.

20.6. File positions

Transcoded ports do not always support the `port-position` and `set-port-position!` operations: The position of a transcoded port may not be well-defined, and may be hard to calculate even when defined, especially when transcoding is buffered.

20.7. Freshness of standard ports

The ports returned by `standard-input-port`, `standard-output-port`, and `standard-error-port` are fresh so it can be safely closed or converted to a textual port without risking the usability of an existing port.

20.8. Argument conventions

While the `(rnrs io simple (6))` library provides mostly R^{5}RS-compatible procedures for performing textual I/O, the `(rnrs io ports (6))` library uses a different convention for argument ordering. In particular, the port is always the first argument. This enables the use of optional arguments for information about the data to be read or written, such as the range in a bytevector. As this convention is incompatible with the convention of `(rnrs io simple (6))`, corresponding procedures have different names.

21. File system

The `(rnrs files (6))` library provides a minimal set of procedures useful in many programs: The `file-exists?` procedure allows a program to detect the presence of a file if it is going to overwrite it, and `delete-file` allows taking the appropriate action if the old file is no longer useful.

Standardization of procedures that return or pass to another procedure the name of a file is more difficult than standardization of `file-exists?` and `delete-file`, because strings are either awkward or insufficient for representing file names on some platforms, such as Unix and Windows. See section 20.1.

22. Arithmetic

22.1. Fixnums and flonums

Fixnum and flonum arithmetic is already supported by many systems, mainly for efficiency. Standardization

of fixnum and flonum arithmetic increases the portability of code that uses it. Standardizing the precision of fixnum and flonum arithmetic would make it inefficient on some systems, which would defeat its purpose. Therefore, R[6]RS specifies the syntax and much of the semantics of fixnum and flonum arithmetic, but makes the precision implementation-dependent.

Existing implementations employ different implementation strategies for fixnums: Some implement the model specified by R[6]RS (overflows cause exceptions), some implement modular arithmetic (overflows "wrap around"), and others do not handle arithmetic overflows at all. The latter model violates the safety requirement of R[6]RS. In programs that use fixnums instead of generic arithmetic, overflows are typically programming mistakes. The model chosen for R[6]RS has the advantage that such overflows do not get silently converted into meaningless number objects, and that the programs gets notified of the violation through the exception system.

22.2. Bitwise operations

The bitwise operations have been adapted from the operations described in SRFIs 33 [31] and 60 [20].

22.3. Notes on individual procedures

`fx+` **and** `fx*` These procedures are restricted to two arguments, because their generalizations to three or more arguments would require precision proportional to the number of arguments.

`real->flonum` This procedure is necessary, because not all real number objects are inexact, and because some inexact real number objects may not be flonums.

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

`flsqrt` The behavior of `flsqrt` on -0.0 is consistent with the IEEE floating-point standard.

23. `syntax-case`

While many syntax transformers are succinctly expressed using the high-level `syntax-rules` form, others cannot be succinctly expressed. Still others are impossible to write, including transformers that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form, transformers that maintain state or read from the file system, and transformers that construct new identifiers. The `syntax-case` system [10] allows the programmer to write transformers that perform these sorts of transformations, and arbitrary additional transformations, without sacrificing the default enforcement of hygiene or the high-level pattern-based syntax matching and template-based output construction provided by `syntax-rules` (report section 11.19).

24. Hashtables

24.1. Caching

The specification notes that hashtables are allowed to cache the results of calling the hash function and equivalence function, and that any hashtable operation may call the hash function more than once. Hashtable lookups are often followed by updates, so caching may improve performance. Hashtables are free to change their internal representation at any time, which may result in many calls to the hash function.

24.2. Immutable hashtables

Hashtable references may be less expensive with immutable hashtables. Also, the creator of a hashtable may wish to prevent modifications, particularly by code outside of the creator's control.

24.3. Hash functions

The `make-eq-hashtable` and `make-eqv-hashtable` constructors are designed to hide their hash function. This allows implementations to use the machine address of an object as its hash value, rehashing parts of the table as necessary if a garbage collector moves objects to different addresses.

25. Enumerations

Many procedures in many libraries accept arguments from a finite set, or subsets of a finite set to describe a certain mode of operation, or several flags to describe a mode of operation. Examples in the R[6]RS include the endianness for bytes-object operations, and file and buffering modes in the I/O library. Offering a default policy for dealing with such values fosters portable and readable code, much as records do for compound values, or multiple values for procedures computing several values. Moreover, representations of sets from a finite set of options should offer the standard set operations, as they tend to occur in practice. One such set operation is the complement, which makes lists of symbols a less than suitable representation.

Different Scheme implementations have taken different approaches to this problem in the past, which suggests that

a default policy does more than merely encode what any sensible programmer would do anyway. As possible uses occur quite frequently, this particular aspect of interface construction has been standardized.

26. Composite library

The `(rnrs (6))` library is intended as a convenient import for libraries where fine control over imported bindings is not necessary or desirable. The `(rnrs (6))` library exports all bindings for `expand` as well as `run` so that it is convenient for writing `syntax-case` macros as well as runtime code.

The `(rnrs (6))` library does not include a few select libraries:

- `(rnrs eval (6))`, as its presence may make creating self-contained programs more difficult;
- `(rnrs mutable-pairs (6))`, as its absence from a program may enable compiler optimizations, and as mutable pairs might be deprecated in the future;
- `(rnrs mutable-strings (6))`, for similar reasons as for `(rnrs mutable-pairs (6))`;
- `(rnrs r5rs (6))`, as its features are deprecated.

27. Mutable pairs

The presence of mutable pairs causes numerous problems:

- It complicates the specification of higher-order procedures that operate on lists.
- It inhibits certain compiler optimizations such as deforestation.
- It complicates reasoning about programs that use lists.
- It complicates the implementation of procedures that accept variable numbers of arguments.

However, removing mutable pairs from the language entirely would have caused significant compatibility problems for existing code. As a compromise, the `set-car!` and `set-cdr!` procedures were moved to a separate library. This facilitates statically determining if a program ever mutates pairs, encourages writing programs that do not mutate pairs, and may help deprecating or removing mutable pairs in the future.

28. Mutable strings

The presence of mutable strings causes problems similar to some of the problems caused by the presence of mutable pairs. Hence, the same reasoning applies for moving the mutation operations into a separate library.

REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., second edition, 1996.

[2] Will Clinger, R. Kent Dybvig, Michael Sperber, and Anton van Straaten. SRFI 76: R6RS records. `http://srfi.schemers.org/srfi-76/`, 2005.

[3] William Clinger. The revised revised report on Scheme, or an uncommon Lisp. Technical Report MIT Artificial Intelligence Memo 848, MIT, 1985 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[4] William Clinger and Jonathan Rees. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[5] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, IV(3):1–55, July–September 1991.

[6] William D Clinger and Michael Sperber. SRFI 77: Preliminary proposal for R6RS arithmetic. `http://srfi.schemers.org/srfi-77/`, 2005.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[8] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, Cambridge, third edition, 2003. `http://www.scheme.com/tspl3/`.

[9] R. Kent Dybvig. SRFI 93: R6RS `syntax-case` macros. `http://srfi.schemers.org/srfi-93/`, 2006.

[10] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[11] Sebastian Egner, Richard Kelsey, and Michael Sperber. Cleaning up the tower: Numbers in Scheme. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the Fifth Workshop on Scheme and Functional Programming*, pages 109–120, Snowbird, October 2004. Indiana University Technical Report TR600.

[12] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. *Scheme 311 version 4 reference manual*. Indiana University, 1983. Indiana University Computer Science Technical Report 137, Superseded by [15].

[13] Matthew Flatt and Kent Dybvig. SRFI 83: R6RS library syntax. `http://srfi.schemers.org/srfi-83/`, 2005.

[14] Matthew Flatt and Marc Feeley. SRFI 75: R6RS unicode data. `http://srfi.schemers.org/srfi-75/`, 2005.

[15] 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.

[16] Lars T Hansen. SRFI 11: Syntax for receiving multiple values. `http://srfi.schemers.org/srfi-11/`, 2000.

[17] C. A. R. Hoare. Algorithm 63 (partition); 64 (quicksort); 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[18] IEEE standard 754-1985. IEEE standard for binary floating-point arithmetic, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[19] IEEE 754 revision work. `http://grouper.ieee.org/groups/754/revision.html`, 2006.

[20] Aubrey Jaffer. SRFI 60: Integers as bits. `http://srfi.schemers.org/srfi-60/`, 2005.

[21] 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.

[22] Richard Kelsey and Michael Sperber. SRFI 34: Exception handling for programs. `http://srfi.schemers.org/srfi-34/`, 2002.

[23] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[24] Jacob Matthews and Robert Bruce Findler. An operational semantics for Scheme. *Journal of Functional Programming*, 2007. From `http://www.cambridge.org/journals/JFP/`.

[25] MIT Department of Electrical Engineering and Computer Science. *Scheme manual, seventh edition*, September 1984.

[26] Paul Penfield Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings*, pages 248–256, San Francisco, September 1981. ACM SIGAPL. Proceedings published as *APL Quote Quad* 12(1).

[27] Jonathan A. Rees and Norman I. Adams IV. T: a dialect of lisp or lambda: The ultimate software tool. In *ACM Conference on Lisp and Functional Programming*, pages 114–122, Pittsburgh, Pennsylvania, 1982. ACM Press.

[28] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, July 1972.

[29] Scheme standardization charter. `http://www.schemers.org/Documents/Standards/Charter/mar-2006.txt`, March 2006.

[30] Olin Shivers. SRFI 1: List library. `http://srfi.schemers.org/srfi-1/`, 1999.

[31] Olin Shivers. SRFI 33: Integer bitwise-operation library. `http://srfi.schemers.org/srfi-33/`, 2002.

[32] Michael Sperber. SRFI 74: Octet-addressed binary blocks. `http://srfi.schemers.org/srfi-74/`, 2005.

[33] 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.

[34] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, and Jonathan Rees. Revised[6] report on the algorithmic language Scheme. `http://www.r6rs.org/`, 2007.

[35] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, and Jonathan Rees. Revised[6] report on the algorithmic language Scheme (Libraries). `http://www.r6rs.org/`, 2007.

[36] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. Technical Report MIT Artificial Intelligence Laboratory Technical Report 474, MIT, May 1978.

[37] Guy Lewis Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, MA, second edition, 1990.

[38] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. Technical Report MIT Artificial Intelligence Memo 452, MIT, January 1978.

[39] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. Technical Report MIT Artificial Intelligence Memo 349, MIT, December 1975.