Revised$^{5.96}$ Report on the Algorithmic Language Scheme
— Non-Normative Appendices —

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

27 June 2007

SUMMARY

This document contains non-normative appendices to the *Revised*^{6} *Report on the Algorithmic Language Scheme*. These appendices contain advice for users and suggestions for implementors on issues not fit for standardization, in particular on platform-specific issues.
This document frequently refers back to the *Revised*^{6} *Report on the Algorithmic Language Scheme* [3] and the *Revised*^{6} *Report on the Algorithmic Language Scheme — Libraries —* [4]; references to the report are identified by designations such as "report section" or "report chapter", and references to the library report are identified by designations such as "library section" or "library chapter".

CONTENTS

*** DRAFT***

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

Appendix A. Standard-conformant mode

Scheme implementations compliant with the report may operate in a variety of modes. In particular, in addition to one or more modes conformant with the requirements of the report, an implementation might offer non-conformant modes. These modes are by nature implementation-specific, and may differ in the language and available libraries. In particular, non-conformant language extensions may be available, including unsafe libraries or otherwise unsafe features, and the semantics of the language may differ from the semantics described in the report. Moreover, the default mode offered by a Scheme implementation may be non-conformant, and such a Scheme implementation may require special settings or declarations to enter the report-conformant mode. Implementors are encouraged to clearly document the nature of the default mode and how to enter a report-conformant mode.

Appendix B. Optional case insensitivity

In contrast with earlier revisions of the report [2], the syntax of data distinguishes upper and lower case in identifiers and in characters specified via their names. For example, the identifiers `X` and `x` are different, and the character `#\space` cannot be written `#\SPACE`.

Implementors may wish to support case-insensitive syntax for backward compatibility or other reasons. If they do so, they are encouraged to adopt the following directives to control case folding.

```
#!fold-case
#!no-fold-case
```

These directives may appear anywhere comments may appear and are treated as comments, except that they affect the reading of subsequent lexemes. The `#!fold-case` causes the reader to case-fold (see library section 1.2) each ⟨identifier⟩ and ⟨character name⟩. The `#!no-fold-case` directive causes the reader to return to the default, non-folding behavior.

Appendix C. Use of square brackets

Even though matched square brackets are synonymous with parentheses in the syntax, many programmers use square brackets only in a few select places. In particular, programmers are encouraged to restrict use of square brackets to places in syntactic forms where two consecutive open parentheses would otherwise be common. These are the applicable forms specified in the report and the library report:

- For `cond` forms (see report section 11.4.5), a ⟨cond clause⟩ may take one of the follow forms:

  `[`⟨test⟩ ⟨expression$_1$⟩ ...`]`
  `[`⟨test⟩ `=>` ⟨expression⟩`]`
  `[else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`]`

- For `case` forms (see report section 11.4.5), a ⟨case clause⟩ may take one of the follow forms:

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

- For `let`, `let*`, `letrec`, `letrec*` forms (see report section 11.4.6), ⟨bindings⟩ may take the following form:

  `(`[⟨variable$_1$⟩ ⟨init$_1$⟩`]` ...`)`

- For `let-values` and `let-values*` forms (see report section 11.4.6), ⟨mv-bindings⟩ may take the following form:

  `(`[⟨formals$_1$⟩ ⟨init$_1$⟩`]` ...`)`

- For `syntax-rules` forms (see report section 11.19), a ⟨syntax rule⟩ may take the following form:

  `[`⟨srpattern⟩ ⟨template⟩`]`

- For `identifier-syntax` forms (see report section 11.19), the two clauses may take the following form:

  `[`⟨id$_1$⟩ ⟨template$_1$⟩`]`
  `[(set!` ⟨id$_2$⟩ ⟨pattern⟩`)` ⟨template$_2$⟩`]`

- For `do` forms (see library section 5), the variable bindings may take the following form:

  `(`[⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩`]` ...`)`

- For `case-lambda` forms (see library section 5), a ⟨case-lambda clause⟩ may take the following form:

  `[`⟨formals⟩ ⟨body⟩`]`

- For `raise` forms (see library section 7.1), a ⟨cond clause⟩ may take one of the follow forms:

  `[`⟨test⟩ ⟨expression$_1$⟩ ...`]`
  `[`⟨test⟩ `=>` ⟨expression⟩`]`
  `[else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`]`

- For `syntax-case` forms (see library chapter 12.4), a ⟨syntax-case rule⟩ may take one of the following forms:

  `[`⟨pattern⟩ ⟨output expression⟩`]`
  `[`⟨pattern⟩ ⟨fender⟩ ⟨output expression⟩`]`

Appendix D. Scripts

A *Scheme script* is a top-level program (see report chapter 8) which is packaged such that it is directly executable by conforming implementations of Scheme, on one or more plaforms.

D.1. Script interpreter

Where applicable, implementations should provide a *script interpreter* in the form of an executable program named `scheme-script` that is capable of initiating the execution of Scheme scripts, as described below.

Rationale: Distributing a Scheme program that is portable with respect to both Scheme implementations and operating systems is challenging, even if that program has been written in standard Scheme. Languages with a single or primary implementation can at least rely on a standard name for their script interpreters. Standardizing the name of the executable used to start a Scheme script removes one barrier to the distribution of Scheme scripts.

D.2. Syntax

A Scheme script is a delimited piece of text, typically a file, with the following syntax:

⟨script⟩ ⟶ ⟨script header⟩ ⟨top-level program⟩
    | ⟨top-level program⟩
⟨script header⟩ ⟶ ⟨shebang⟩ `/usr/bin/env` ⟨space⟩
    `scheme-script` ⟨linefeed⟩
⟨shebang⟩ ⟶ `#!` | `#!` ⟨space⟩

D.2.1. Script header

The *script header*, if present on the first line of a script, is used by Unix-like operating systems to identify the interpreter to execute that script.

The script header syntax given above is the recommended portable form that programmers should use. However, if the first line of a script begins with `#!/` or `#!`⟨space⟩, implementations should ignore it on all platforms, even if it does not conform to the recommended syntax.

Rationale: Requiring script interpreters to recognize and ignore the script header helps ensure that Scheme scripts written for Unix-like systems can also run on other kinds of systems. Furthermore, recognizing all `#!/` or `#!`⟨space⟩ combinations allows local customizations to be performed by altering a script header from its default form.

D.2.2. Example

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

D.3. Platform considerations

Many platforms require that scripts be marked as executable in some way. The platform-specific details of this are beyond the scope of this report. Scripts that are not suitably marked as executable will fail to execute on many platforms. Other platform-specific notes for some popular operating systems follow.

D.3.1. Apple Mac OS X

The Mac OS X operating system supports the Unix-like script header for shell scripts that run in the Terminal. Depending on the intended usage, it may be advisable to choose a file name ending in `.command` for a script, as this makes the script double-clickable.

D.3.2. Unix

Scheme scripts on Unix-like operating systems are supported by the presence of the script header. Scripts that omit the script header are unlikely to be directly executable on Unix-like systems.

To support installation of the Scheme script interpreter in non-standard paths, scripts should use the `/usr/bin/env` program as specified in the recommended script header syntax. (Note that on many Unix-like systems, this also allows the script interpreter itself to be implemented as a shell script.)

D.3.3. Microsoft Windows

The Windows operating system allows a file extension to be associated with a script interpreter such as `scheme-script`. This association may be configured appropriately by Scheme implementations, installation programs, or by the user.

D.3.4. Selecting an implementation

If multiple implementations of Scheme are installed on a machine, the user may wish to specify which implementation should be used to execute Scheme scripts by default. Most platforms support some mechanism for choosing between alternative implementations of a program. For example, Debian GNU/Linux uses the `/etc/alternatives`

mechanism to do this; Microsoft Windows uses file extension associations. Implementations are expected to configure this appropriately, e.g., as part of their installation procedure. Failing that, users must perform any necessary configuration to choose their preferred Scheme script interpreter.

Appendix E. Source code representation

The report does not specify how source code is represented and stored. The only requirement the report imposes is that the source code of a top-level program (see report section 8.1) or a script (see section D.2) be delimited. The source code of a library is self-delimiting in the sense that, if the beginning of a library form can be identified, so can the end.

Implementations might take radically different approaches to storing source code for libraries, among them: files in the file system where each file contains an arbitrary number of library forms, files in the file system where each file contains exactly one library form, records in a database, and data structures in memory.

Similarly, programs and scripts might be stored in a variety of formats. Platform constraints might restrict the choices available to an implementation, which is why the report neither mandates nor recommends a specific method for storage.

Implementations may provide a means for importing libraries coming from the outside via an interface that accepts a UTF-8 text file in Unicode Normalization Form C where line endings are encoded as newline characters. Such text files may contain an arbitrary number of library forms. (Authors of such files are encouraged to include an `#!r6rs` comment if the file is written purely with the lexical and datum syntax described in the report. See report section 4.2.3.) After importing such a file, the libraries defined in it should be available to other libraries and files. An implementation may store the file as is, or convert it to some storage format to achieve this.

Similarly, implementations may provide a means for executing a program represented as a UTF-8 text file containing its source code. Again, authors of such files are encouraged to include an `#!r6rs` comment if the file is written purely with the lexical and datum syntax described in the report. This report does not describe a file format that allows both libraries and programs to appear in the same file.

Appendix F. Use of library versions

Names for libraries may include a version. An ⟨import spec⟩ may designate a set of acceptable versions that may be imported. Conversely, only one version of each library should be part of a program. This allows using the "name part" of a ⟨library name⟩ for different purposes than the ⟨version⟩.

In particular, if several different variants of a library exists where it is feasible that they coexist in the same program, it is recommended that different names be used for the variants. In contrast, for compatible versions of a library where coexistence of several versions is unnecessary and undesirable, it is recommended that the same name and different versions be used. In particular, it is recommended new versions of libraries that are conservative extensions of old ones differ only in the version, not in the name.

Correspondingly, it is recommended that ⟨import spec⟩s do not constrain an import to a single version, but instead specify a wide range of acceptable versions of a library.

Implementations that allow two libraries of the same name with different versions to coexist in the same program are encouraged to report when processing a program that actually makes use of this extension.

Appendix G. Unique library names

Programmers are encouraged to choose names for distributed libraries whose names are chosen not to collide with other libraries' names. This appendix suggests a convention for generating unique library names, similar to the convention for Java [1].

A unique library name can be formed by associating the library with an Internet domain name, such as `mit.edu`. The lower-case components of the domain are reversed to form a prefix for the library name. Adding further name components to establish a hierarchy may be advisable, depending on the size of the organization associated with the domain name, the number of libraries to be distributed from it, and other organizational properties or conventions associated with the library.

Programmers are encouraged to use library names that are suitable for use in the file-system mapping described in appendix H. Special characters in domain names that do not fit the convention should be replaced by hyphens or suitable "escape sequences" that, as much as possible, are suitable for avoiding collisions. Here are some examples for possible library names according to this convention:

```
(edu mit swiss cheese)
(de deinprogramm educational graphics turtle)
(com pan-am booking passenger)
```

The name of a library does not necessarily indicate an Internet address where the package is distributed.

Appendix H. Library / file system mapping

Storing library source code as files in a hierarchical file system is a common way to support the use of standard tools for editing and other kinds of source code processing.

The following recommendation specifies a standard way to map library names to file names in widely-used file systems, using an approach in which each file contains exactly one library form. Following this recommendation will allow users to work with a familiar source code structure across implementations, and can also allow multiple implementations to share a common repository of library source code.

The form of a library name is specified in section 7.1. It can be expressed as follows:

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

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

(⟨sub-version$_1$⟩ ⟨sub-version$_2$⟩ ...)

Such a library name should be mapped to a file in the file system with a relative path formed by the concatenation of the following components:

⟨identifier$_1$⟩ ⟨sep⟩ ... ⟨identifier$_n$⟩ ⟨xsep⟩ ⟨extension⟩

where ⟨sep⟩ is the platform-specific character (such as `/`) used to separate path elements (which are typically directory names); ⟨xsep⟩ is the platform-specific character (typically a period) used to separate parts of a file name; and ⟨extension⟩ has the following form:

⟨sub-version$_1$⟩ ⟨xsep⟩ ⟨sub-version$_2$⟩ ⟨xsep⟩ `... sls`

where `sls` is the recommended extension used to identify Scheme library source files.

Note that the resulting path is relative to some implementation-dependent root directory.

According to this mapping, the source code for a library named `(mylib examples hello)` would be stored in a file `mylib/examples/hello.sls`; the source code for a library named `(mylib examples hello (0 4 2))` would be stored in a file `mylib/examples/hello.0.4.2.sls`.

A library source file may define a library with a library name consisting of the same sequence of identifiers as another library known to the implementation if each library name includes a distinct and non-empty ⟨version⟩.

If a library source file defines a library with a library name for which ⟨version⟩ is `()` or empty, then the source file must similarly have no version embedded within its name. In that case, to avoid confusion, no other library with a library name consisting of the same sequence of identifiers, but with a non-empty version, should be known to the implementation.

This report does not describe a file system mapping for compiled code.

REFERENCES

[1] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, third edition, 2005.

[2] 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.

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

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