# Implementing R7RS on an R6RS Scheme system

Takashi Kato
Bell ID B.V.
t.kato@bellid.com

## Abstract

The Scheme language has three major standards; Revised$^5$ Report on the Algorithmic language Scheme (R5RS) standardised in February 1998, the Revised$^6$ Report on Algorithmic language Scheme (R6RS) standardised in September 2007 and the Revised$^7$ Report on the Algorithmic language Scheme (R7RS) standardised in July 2013. R7RS, the latest standard of Scheme focuses on the R5RS compatibility thus making R5RS implementations compliant with it would not be so difficult. For R6RS implementations it would be much more difficult; R7RS clearly says it is not a successor of the R6RS. This paper describes the major differences between these two Scheme standards and how we made our Scheme system, Sagittarius, compliant with both R6RS and R7RS, and made it able to use both standards' libraries seamlessly.

***Keywords*** Scheme, R6RS, R7RS

## 1. Introduction

The Revised$^6$ Report on Algorithmic language Scheme (R6RS) [2] was completed in September 2007 with many new improvements and a focus on portability. Some implementations were adopted for R6RS. Some R6RS compliant implementations were created. In July 2013, The Revised$^7$ Report on Algorithmic language Scheme (R7RS) [3] was completed with the focus on the Revised$^5$ Report on Algorithmic language Scheme (R5RS) [1] compatibility. Both R6RS and R7RS are R5RS compatible, however these two standards are not totally compatible. Therefore, these two standards are not able to share libraries nor scripts.

We have searched repositories on GitHub and Google Code with keyword "R6RS" and "R7RS", and repository language "Scheme" in August 2014. On GitHub, there were 59 repositories related to R6RS and 12 repositories related to R7RS. On Google Code, there were 18 repositories related to R6RS and 8 repositories related to R7RS.

Table 1: Number of Repositories

| Keyword | GitHub | Google Code |
|---|---|---|
| R6RS | 59 | 18 |
| R7RS | 12 | 8 |

The search result may contain implementations themselves and may not contain repositories which do not have the keywords in their description or searchable locations. So these are not accurate numbers of repositories that provide libraries. However, it has only been one year since R7RS standardised so we can expect the numbers of R7RS repositories to grow in near future. We have concluded that it is important to support the R7RS on our Scheme system, Sagittarius[1] which base is R6RS, so that it would be beneficial for future Scheme users. One of our goals is using R6RS libraries in R7RS library form and vice versa. The following sections describe how we implemented the R7RS on top of the R6RS Scheme system and how both R6RS and R7RS libraries can inter-operate.

## 2. Incompatibilities

R7RS lists numerous incompatibilities with R6RS. However, incompatibilities of procedures or macros are negligible because both R6RS and R7RS support renaming import and export. So we only need to define them for R7RS libraries. For example, the R6RS `let-syntax` must be sliced into `begin` however the R7RS one must create a scope. If an implementation has the R6RS style `let-syntax`, then it is easy to implement the R7RS style one with it. A possible implementation of the R7RS style `let-syntax` would look something like the following:

Listing 1: R7RS style let-syntax

```
;; R7RS style let-syntax
(import
  (rename (rnrs)
          (let-syntax r6rs:let-syntax)))
(define-syntax let-syntax
  (syntax-rules ()
    ((_ ((vars trans) ...) expr ...)
     (r6rs:let-syntax ((vars trans) ...)
       (let () expr ...)))))
```

Thus, the incompatibilities we need to discuss here are the layers that require deeper support of implementations such as library forms and lexical notations.

### 2.1 Library forms

A library or module system is essential for modern programming languages to allow programmers to reuse useful programs. However, the Scheme language did not provide this until the R6RS was standardised. R6RS decided to call it a library system so we also call it library system here. From Scheme language perspective, it is quite a new concept. R7RS has also been standardised with a library system however it does not have the same form as the R6RS.

The R6RS has `library` keyword for library system whilst the R7RS has `define-library`. The R6RS does not define

[1] Sagittarius Scheme: https://bitbucket.org/ktakashi/sagittarius-scheme/

mechanism to absorb incompatibilities between implementations nor to check whether required libraries exist. Thus making a portable library requires using unwritten rules. The library system of the R7RS, on the other hand, does have the feature provided by `cond-expand` keyword.

To demonstrate the difference between R6RS and R7RS library forms, we show concrete examples of both. The library `foo` exports the variable *bar* and requires an implementation dependent procedure.

### 2.1.1 R6RS library

The R6RS library system has rather fixed form. With the R6RS `library` form, the library `(foo)` would look like the following:

Listing 2: R6RS library form

```
(library (foo)
    (export bar)
    (import (rnrs) (compat foo))
  (define bar (compat-foo-proc)))
```

An R6RS library name can only contain symbols and a version reference at the end of library name[2], which must be a list of numbers. Both `export` and `import` forms must be present only once in respective order.

Here, the `(compat foo)` is a compatible layer of an implementation-dependent procedure. R6RS does not have the means to load implementation-specific code, however, there is a *de-facto* standard supported by most of the R6RS implementations listed on `http://www.r6rs.org/implementations.html`. If the implementation is Sagittarius Scheme, for example, then the filename of its `(compat foo)` library would be `compat/foo.sagittarius.sls` which could contain something like the following:

Listing 3: Compatible layer

```
;; compat/foo.sagittarius.sls
(library (compat foo)
    (export compat-foo-proc)
    (import (sagittarius))
  (define compat-foo-proc
    implementation-dependent-procedure))
```

If the library wants to provide a default procedure, then it needs to have `compat/foo.sls` as the default library file name. The implementations first try to resolve the library file name with its featured name then fall back to the default filename. This requires the same number of compatible layer library files as implementations that the library would support. Moreover, it is not guaranteed to be portable by the standard.

### 2.1.2 R7RS define-library

The R7RS library system provides much more flexibility than the R6RS library system does. With the R7RS `define-library` form, the `(foo)` library can be written something like the following:

Listing 4: R7RS define-library form

```
(define-library (foo)
  (import (scheme base))
  (cond-expand
    ((library (bar))
     (import (bar)))
    (sagittarius
     (import (sagittarius))
     (define bar
       implementation-dependent-procedure))
    (else
     (error "unsupported implementation")))
  (export bar))
```

[2] Version reference is optional and it is meant that user can choose a specific version of using library. However, most of the implementations ignore it.

An R7RS library name can contain symbols and numbers, and does not support library version references. Thus, `(srfi 1)` is a valid library name whilst the R6RS one needs to be written something like `(srfi :1)`.

Moreover, unlike the R6RS `library` form, R7RS supports more keywords, `import`, `export`, `cond-expand`, `include`, `include-ci` and `include-library-declarations`. Using `cond-expand` makes the R7RS library system enables writing implementation-dependent code without separating library files.

The above example does not show, however, using `include` or `include-ci`, which enable including files from outside of the file where libraries are defined. And `include-library-declarations` includes files containing library declarations.

### 2.1.3 Export form

Besides those overall differences, the R6RS and R7RS have slightly different syntax for the `rename` clause of `export` forms. The R6RS `export` may have multiple renamed exporting identifiers whilst the R7RS `export` only allows to have one renamed exporting identifier. So the R7RS form requires multiple `rename` clauses to export more than one identifier with different names.

Listing 5: R6RS export

```
(export (rename (foo foo:foo)
                (bar foo:bar)))
```

Listing 6: R7RS export

```
(export (rename foo foo:foo)
        (rename bar foo:bar))
```

## 2.2 Lexical incompatibilities

Basic lexical representations for data types are shared between R6RS and R7RS. However, the symbol escaping and the bytevector notation from R6RS have been changed in R7RS[3].

### 2.2.1 Symbols

A lot of R5RS implementations have a relaxed symbol reader that allows symbols to start with "`@`" or "`.`" which R5RS does not allow[4]. And some of de-facto standard libraries, such as SXML [4], depend on it. However, R6RS does not allow identifiers to start with these characters and mandates implementations to raise an error. So writing those symbols requires escaping like the following:

[3] Additionally, the R7RS supports shared data structure notations however it is an error if program or library form contains it. Thus, only the `read` procedure needs to support it and it can be defined in R7RS library. So we do not discuss it here.

[4] In R5RS, "it is an error" means implementations do not have to raise an error so they may allow them as their extension. The same rule is applied to R7RS. The R6RS has strict error condition. It specifies that which condition implementations must raise.

Listing 7: R6RS symbol escaping

```
\x2E;foo ;; -> .foo
\x40;bar ;; -> @bar
```

This does not break R5RS compatibility however it does break de-facto standards and most R6RS implementations adopt the strict lexical rule[5]. Therefore, non-R5RS symbols cannot be read by these implementations.

R7RS has decided to allow those symbols so that implementations can use R5RS libraries without changing code. R7RS also supports symbol escaping using vertical bars "`|`". Hex scalar, the same as R6RS supports, is also allowed inside of vertical bars. The R7RS escaped symbol notation would look something like the followings:

Listing 8: R7RS symbol escaping

```
|foo bar|      ;; -> |foo bar|
|foo\x40;bar| ;; -> |foo@bar|
```

Hex escaped symbols are not required to be printed with hex scalar even if the value is not a printable character such as "U+007F".

Unlike the R6RS, the R7RS hex escaping can only appear inside of vertical bars[6]. Thus the two standards do not share the escaped symbol notations.

### 2.2.2 Bytevectors

Since R6RS, Scheme can handle binary data structure called bytevectors. The data structures can contain octet values which are exact integers ranging from 0 to $2^8 - 1$. Both standards support it however the lexical notations are not the same. There is a Scheme Requests For Implementation (SRFI) for binary data types, SRFI 4: Homogeneous numeric vector datatypes [5]. With this SRFI, the binary data types are similar to bytevectors and can be written like the following:

Listing 9: u8vector

```
#u8(0 1 255)
```

The SRFI defines more data types and their external representation such as 32 bit integer vectors. It also defines procedures such as getters and setters.

R6RS has adopted its concept, but has not taken the name and the external representation as it is. Instead, writing a bytevector literal in R6RS looks like the following:

Listing 10: R6RS bytevector

```
#vu8(1 2 3)
```

To handle the other data types defined in the SRFI, the R6RS provides conversion procedures which can treat a bytevector as if it is a vector of other data type such as 32 bit integer. Take as examples `bytevector-u32-ref` and `bytevector-u32-set!`. The first one can retrieve a 32 bit integer value from a bytevector and the second one can set a 32 bit integer value into a bytevector.

R7RS, on the other hand, has decided to use the SRFI as it is but only the octet values one. The lexical notation of R7RS bytevector is the same as SRFI 4. Even though it has only one type of bytevector, there is no conversion procedure provided.

[5] Some implementations have strict reader mode and compatible mode.

[6] Initially, the R7RS had both vertical bar notation and the R6RS style hex scalar notation. But the R6RS compatible notation was removed. http://trac.sacrideo.us/wg/ticket/304

## 3. Implemention strategy

There are several strategies to implement R7RS on R6RS. Here we discuss handling different library forms and lexical notations.

### 3.1 Expander vs built-in

There are two portable R6RS expanders which provide the R6RS library system, `syntax-case` and some procedures and the macros. One is SRFI 72: Hygienic macros [6] and the other one is Portable syntax-case (psyntax) [7]. These expanders pre-process and expand libraries and macros. Knowing this gives us two possible solutions to implement the R7RS library system. One is to build the R7RS library system on top of the R6RS library system by transforming the R7RS `define-library` form to the R6RS `library` form like these expanders do. We call this expander style. The other one is for implementations to support the library form as their built-in keyword. We call this built-in style. There are advantages and disadvantages for both strategies.

Built-in style requires changing expanders or compilers. Thus, it is the more difficult method to implement. However, it give us more control so that it has the same expansion phase of existing library systems. Thus, during a library compilation, it can refer the same compile time environment as the expanders can.

Expander style is, on the other hand, easier to implement and can keep the portable code intact. However, it may impact the performance of loading libraries. It first needs to transform `define-library` forms to `library` forms then underlying R6RS expanders expand library forms and macros. Moreover, transforming library forms may introduce phasing issues. Phasing has been introduced for the R6RS library system with keyword `for` to resolve macro-expansion time environment references. Psyntax implicitly resolves the phase but the SRFI 72 expander mandates explicit phasing. However, R7RS does not specify phasing because it has only `syntax-rules` as its macro transformer and it does not require phasing. It depends on underlying R6RS expanders, however: the library form transformer would need to consider in which phase imported libraries are used. Since R7RS does not require phasing, the only case it would be a problem is that of procedural macros used in R7RS libraries. For example, suppose we have the following R7RS library form.

Listing 11: Phasing

```
(define-library (foo)
  (import (rnrs))
  (begin
    (define-syntax foo
      (lambda (x)
        (define-syntax name
          (syntax-rules ()
            ((_ k)
             (datum->syntax k
               (string->symbol "bar")))))
        (syntax-case x ()
          ((k)
           (with-syntax ((def (name #'k)))
             #'(define def 'bar)))))))
  (export foo))
```

If underlying R6RS expanders have explicit phasing, then the transformation of the `define-library` form to a `library` form would need to traverse the macro `foo` to detect which phase it requires. And it needs to add proper indication of the required phase. One of the possible transformation results would be the following:

Listing 12: Possible transformation

```
(library (foo)
    (export foo)
    (import (for (rnrs) run expand))
  (define-syntax foo
    (lambda (x)
      (define-syntax name
        (syntax-rules ()
          ((_ k)
           (datum->syntax k
             (string->symbol "bar")))))
      (syntax-case x ()
        ((k)
         (with-syntax ((def (name #'k)))
           #'(define def 'bar)))))))
```

Besides the phasing issue, R7RS also requires "include" mechanism as one of the keywords inside of `define-library` and syntax. And this requires implementations to properly resolve file paths. Suppose library `foo` includes "impl/bar.scm" which itself includes "buzz.scm". R7RS actually does not specify how this nested include should be resolved however is seems natural that the `include` form in "impl/bar.scm" should include "impl/buzz.scm" just as the C's `#include` preprocessor which resolves an included file's location from where its includer is located[7].

Listing 13: Nested include

```
#|
File hierarchy
 /
  + foo.sld
    + impl/
      + bar.scm
      + buzz.scm
    + buzz.scm
|#
;; foo.sld
(define-library (foo)
    (import (scheme base))
    (export bar)
  (include "impl/bar.scm"))

;; impl/bar.scm
(include "buzz.scm")

;; impl/buzz.scm
(define bar 'bar)

;; buzz.scm
(define bar 'boo)
```

Suppose we have two files "buzz.scm": one is inside of "impl" directory and the other is in the same directory as "foo.sld" is located. "impl/buzz" and "buzz.scm" define a binding `bar` which has values `bar` and `boo`, respectively. And a library `foo` exports the binding `bar`. If implementations resolve this as the C's `#include` preprocessor does, then the bound value of `bar` would be a symbol `bar`. However, if it does not, then it would be a symbol `boo`.

Implementing such a behaviour requires meta information of source file locations and expression mappings, so R7RS library expanders need to know where expressions are read from. Thus, the expanders are required to traverse transforming expressions and find `include` expressions to include nested inclusion properly. However, finding these expressions also requires the analysis of bindings. If the syntax `include` is shadowed or not imported, then the expander should not resolve it as an `include` expression but a mere symbol. Therefore, it also needs to have binding environment managing which the R6RS expander does. Moreover, if a macro contains an `include` expression, this would also be hard to implement in expander-style.

[7] Implementations may decide to implement complete opposite way, that is discouraging users to use nested include or include-ci syntax.

Listing 14: Macro with include

```
(define-syntax include-it
  (syntax-rules ()
    ((_ file) (include file))))
```

In this case, the macro could be expanded anywhere and the file location would depend on where it is expanded. Thus, `define-library` expanders need to handle macros during transforming so that they can resolve file locations properly.

## 3.2 Reader and writer modes

As we discussed, R6RS and R7RS have different symbol escaping styles and lexical notations for bytevectors. It is not difficult to support reading; supporting writing is more challenging. One of the specific advantage of Lisp dialect languages is the read and write invariance. Thus writing them in expected form is necessary.

One solution is to use `#!`. R6RS has the `#!r6rs` notation so if a script has this, then implementations can choose R6RS style writer. R7RS, on the other hand, does not define `#!r7rs` notation and if implementations choose to strictly adhere to R7RS then this would be an error. Therefore, switching reader or writer mode by `#!` notation only works for R6RS scripts. Thus using `#!` notation to switch mode without depending on implementation-specific features requires the default mode of the reader and writer to be R7RS.

Another solution is to detect library forms. When the reader find `define-library` form, then it should switch to R7RS mode. Doing this requires two-pass reading since a library form is one S-expression. First the reader reads one expression and checks whether or not it is a list whose first element is a `define-library` symbol. If it is, then the reader needs to discard the expression and re-reads it with R7RS mode. This only works for loading libraries and reading expressions, and requires the reader to be able to handle positioning. Writing the R7RS style symbols and bytevectors requires something else.

Switching mode only works if reading and writing are done by only one Scheme implementation. If more than one implementation needs to share code or written S-expressions, then it will be a problem. Suppose a server-client type application is running on three implementations. The server is an R6RS and R7RS compliant implementation and one of the clients is R6RS compliant, the other client is R7RS compliant. Now the data exchange is done with S-expressions so that all implementations can use the bare `read` procedure. However, if the data being exchanged can also contain bytevector, the server would not be able to determine which style of bytevector form it should send. Unless, that is, the exchanging data contains a client mode so that the server can detect which style of notation it should use. This problem occurs not only for bytevectors but also for escaped symbols.

Listing 15: Example situation

```
;; Server S is a hybrid implementation but
;; would return R6RS style lexical notation.

;; Client A is an R6RS implementation
Client A -- #vu8(1 2 3) --> Server S
Client A <-- #vu8(1 2 3) -- Server S

;; Client B is an R7RS implementation
Client B --  #u8(1 2 3) --> Server S
Client B <-- #vu8(1 2 3) -- Server S
```

## 4. Implementing on Sagittarius

Sagittarius has strict R6RS read and write mode and relaxed mode. The macro expander does not have an explicit macro expansion phase so the compiler expands macros as well when it finds a macro. The mode switch is done by `#!` notation and by default it is set to relaxed mode which is close to R7RS compatible with some extensions.

### 4.1 Library

We decided to implement `define-library` with the built-in style so that macro expansions are done by the existing expander. For the most part, handling the R7RS library form could be implemented in the same style as R6RS library system. However, unlike the R6RS, the R7RS allows all keywords inside of `define-library` such as `import` and `export` to appear in any order and any number of times.

Listing 16: Multiple imports

```
(define-library (foo)
  (begin
    (define bar 'bar)
    (define foo 'foo))
  (import (scheme base))
  (begin (display bar) (newline))
  (import (scheme write))
  (export bar foo))
```

`Import` forms need to be collected before bodies are compiled, otherwise the compiler can not find imported bindings referred by body expressions. For example, if a body expression depends on bindings exported from the library inside of `import` forms which comes after the body expression, then the compiler raises an error. During the process of collecting of `import` forms, the compiler needs to keep the order of `begin` forms so that it can resolve bindings properly when `begin` forms contain non-definition expressions[8].

Even though we have decided to take the built-in style, `include` and `include-ci` need to be handled specially. These are resolved in a hybrid way. The ones in `define-library` are resolved in the expander style, thus when the compiler finds it in a library form, it simply reads the files and slices the expressions into the library form. Syntaxes of `include` and `include-ci` are resolved in the built-in style so that they can be treated as bindings. However, read expressions of both styles contain location information as part of their meta information. This meta information is propagated to compile time environments so that the compiler can see where the source files are located.

The expander style `include` is expanded as it is. The only thing that the compiler needs to consider is propagating the source file locations to the rest of compilation unit.

The built-in style needs to be more careful. Besides the compiler needs to consider bindings. If an `include` form appears in top level, it is relatively easy to handle. However, if it appears in a scope, then the compiler needs to consider lexical bindings. The following is a simple example.

[8] The R7RS allows to have non definition forms anywhere inside of `begin` forms.

Listing 17: Local include

```
(let ((bar 'bar))
  (include "bar.scm")
  buzz)

;; bar.scm
(define buzz 'buzz)
(display bar) (newline)
```

The `define` form inside of "bar.scm" needs to be handled as an internal definition. So the compiler needs to handle `include` forms inside of a scope explicitly otherwise a `define` form would be treated as a toplevel form and the compiler would raise an error. If macros were expanded before compilation with proper source location, this would not be a problem. However, this requires accessing the meta information, and there is no way to do so on our Scheme system.

Resolving `export` is straightforward. There are two ways to do it: one is to implement an R7RS-specific one, and the other one is to make the R6RS `export` able to handle the R7RS style as well. We chose the latter, so that shared code can be used. However, we are not certain that this was the right way to do it yet.

### 4.2 Reader and writer

The reader needs to adopt two incompatibilities with R6RS, one is the escaped symbol and the other one is the bytevector literal. The reader on our Scheme system adopted Common Lisp-like reader macros, thus handling bytevector notation incompatibilities is just adding the additional reader macro. Handling vertical bar-escaped symbols also requires just adding the reader macro. However, when reading usual symbols we need to provide both the R6RS symbol reader and the R7RS symbol reader.

To make strict modes for R6RS and R7RS, the Scheme system has three default readtables which are tables of bundled reader macros. One is the R6RS strict mode, another one is the R7RS strict mode and the last one is the default mode. Switching these readtables requires `#!r6rs` or `#!r7rs` notations. As we already discussed R7RS does not support `#!r7rs`, thus switching mode with this is our specific extension and may break portability.

The writing of escaped symbols and the bytevectors literals is also separated into modes. In the strict R6RS mode, the escaped symbols are written without vertical bar and bytevectors are written with `#vu8` notation. In the strict R7RS mode, bytevectors are written with `#u8` notation. If an escaped symbol contains non-printable characters then they are written in hex scalar. The default relaxed mode can read both the R6RS and the R7RS lexical notations so that it can understand both types of scripts and libraries. Its writer mode is hybrid, escaped symbols are written in R7RS style[9] and bytevectors are written R6RS style.

Listing 18: Read/write symbols and bytevectors

```
;; mode: default
'|foo\x20;bar|  ;; -> '|foo bar|
'foo\x20;bar    ;; -> '|foo bar|

#vu8(1 2 3)     ;; -> #vu8(1 2 3)
#u8(1 2 3)      ;; -> #vu8(1 2 3)

;; mode: R6RS
#!r6rs
```

[9] This is for an historical reason. Sagittarius was initially made as an R5RS/R6RS Scheme system. So we did not have to consider the difference between bytevector lexical notations, and writing R6RS-style escaped symbols in default mode breaks R5RS compatibility.

```
'|foo\x20;bar|  ;; error
'foo\x20;bar    ;; -> 'foo\x20;bar

#vu8(1 2 3)     ;; -> #vu8(1 2 3)
#u8(1 2 3)      ;; error

;; mode: R7RS
#!r7rs
'|foo\x20;bar|  ;; -> |foo bar|
'foo\x20;bar    ;; -> |foo bar|[10]

#vu8(1 2 3)     ;; error
#u8(1 2 3)      ;; -> #u8(1 2 3)
```

## 5. Other R7RS features

We have discussed the major incompatibilities and how we handled them. There are some other points that still need to be considered.

### 5.1 cond-expand

As we already discussed, the R7RS `define-library` form allows the `cond-expand` keyword, which is based on SRFI 0: Feature-based conditional expansion construct [8] with the `library` keyword extension. The `library` keyword allows checking if the specified library exists on the executing implementation.

Listing 19: Library keyword in cond-expand

```
(define-library (foo)
  (cond-expand
    ((library (srfi 1))
     (import (srfi 1)))
    (else
     (begin
       (define (alist-cons a b c)
         (cons (cons a b) c)))))
  (import (scheme base)))
```

The `cond-expand` form inside of a `define-library` form can only have library declarations in its body. There is another `cond-expand` defined as a syntax in R7RS which can be used in expressions. This is close to SRFI 0 but added `library` as a keyword. However, this body can only take expressions thus it is invalid to write an `import` form[11].

Listing 20: cond-expand in expression

```
;; This is not a valid R7RS program
(cond-expand
  ((library (srfi 1))
   (import (srfi 1)))
  (else
   (define (alist-cons a b c)
     (cons (cons a b) c))))
```

Even though this is not a valid program, we decided to accept this type of expressions its support is recommended one of the R7RS editors [9].

[10] It is an extension that`'foo\x20;bar` can be read in strict R7RS mode even thoughtit is defined to be an error.

[11] R7RS defined that `cond-expand` can only have expressions and `import` form is not an expression.

### 5.2 #!fold-case and #!no-fold-case

Like R6RS, R7RS has decided to make symbols case-sensitive. However, until R5RS, the Scheme language was case-insensitive so there may be some scripts or libraries that expect to be case-insensitive. To save such programs, R7RS has introduced the `#!fold-case` directive and the `include-ci` form.

If the reader reads `#!fold-case` then it should read expressions after the directive as case-insensitive, and if it reads `#!no-fold-case`, it should read expressions after the directive as case-sensitive. These directives can appear anywhere in scripts or libraries. Thus, to handle this, ports need to have the state in which they read symbols.

The symbols read in case-insensitive context need to be case folded as if by `string-foldcase`. Thus comparing symbol `ß` and `ss` needs to return `#t` in case-insensitive context[12].

Listing 21: #!fold-case

```
#!fold-case
(eq? 'ß 'ss) ;; => #t
```

## 6. Interoperability

We show how R6RS and R7RS libraries cooperate on our Scheme system.

Suppose we have the library `(aif)` which defines anaphoric if macro with `syntax-case`. The macro `aif` is similar to `if`. The difference is that it captures the variable `it` as the result of its predicate and *then* and *else* forms can refer it. This is a typical macro can not be written in `syntax-rule`.

Listing 22: aif

```
#!r6rs
(library (aif)
    (export aif)
    (import (rnrs))
  (define-syntax aif
    (lambda (x)
      (syntax-case x ()
        ((aif c t) #'(aif c t (if #f #t)))
        ((k c t e)
         (with-syntax
             ((it (datum->syntax #'k 'it)))
           #'(let ((it c))
               (if it t e))))))))
```

The R7RS library `(foo)` defines the variable `foo` using `aif` defined in the R6RS library.

Listing 23: Using aif

```
(define-library (foo)
  (import (scheme base) (aif))
  (export foo)
  (begin
    (define foo
      (let ((lis '((a . 0) (b . 1) (c . 2))))
        (aif (assq 'a lis)
             (cdr it))))))
```

The variable `foo` can be used in user scripts, R7RS libraries or R6RS libraries.

[12] The R6RS mandates to support Unicode so `string-foldcase` does full case folding.

As we already mentioned, Sagittarius has implicit phasing so it is also possible to use procedural macros in R7RS libraries without the `for` keyword.

## 7. Conclusion

We have discussed the incompatibilities between R6RS and R7RS and described implementation strategies. Then we discussed how we built an R7RS Scheme system on top of an R6RS Scheme system. What we have experienced so far is that as long as implementation could absorb those difference, there is no problem using the R6RS library system and the R7RS library system simultaneously. And we believe that this could be a big benefit for the future.

Implementing an R7RS-compliant Scheme system on top of an R6RS Scheme system is not an easy task to do. Moreover, most R6RS users do not habitually use R7RS and vice versa. However, we believe that both standards have good points that are worth taking. We think that complying standards is also important for implementators.

We hope this will encourage R6RS implementators to make their implementation R7RS compliant as well.

## Acknowledgments

We would like to thank Shiro Kawai for style checking and advising.

## References

[1] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised$^5$ Report on the Algorithmic Language Scheme. February 1998.

[2] Michael Sperber, R. Kent Dybvig, Mathew Flatt, and Anton van Straaten, editors. Revised$^6$ report on the algorithmic language Scheme. September 2007.
URL http://www.r6rs.org/

[3] Alex Shinn, John Cowan, and Arthur A. Gleckler, editors. Revised$^7$ report on the algorithmic language Scheme. July 2013.
URL http://scheme-reports.org/

[4] Oleg Kiselyov. SXML Specification. March 2004
URL http://pobox.com/ oleg/ftp/Scheme/SXML.html

[5] Marc Feeley. SRFI 4: Homogeneous numeric vector datatypes. May 1999.
URL http://srfi.schemers.org/srfi-4/

[6] André van Tonder. SRFI 72: Hygienic macros. September 2005.
URL http://srfi.schemers.org/srfi-72/

[7] Abdulaziz Ghuloum and R. Kent Dybvig. Portable syntax-case
URL http://www.cs.indiana.edu/chezscheme/syntax-case/

[8] Marc Feeley. SRFI 0: Feature-based conditional expansion construct. May 1999
URL http://srfi.schemers.org/srfi-0/

[9] [Scheme-reports] programs and cond-expand.
URL http://lists.scheme-reports.org/pipermail/scheme-reports/2013-October/003802.html