Sagittarius Users' Reference

version 0.2.0

This document is a manual for Sagittarius, a Mostly R6RS Scheme implementation. This is for version 0.2.0.

Contents

1 Introduction

This is a users' guide and reference manual of Sagittarius Scheme system. Here I tried to describe the points which are not conformed to R6RS.

The target readers are those who already know Scheme and want to write useful program in Sagittarius.

This manual only deals with Scheme side of things. Sagittarius has another face, a C interface. Details of it will be discussed in a separate document to be written. Those who wants to use Sagittarius as an embedded language, or wants to write an extension, need that volume.

1.1 Overview of Sagittarius

Sagittarius is a Scheme script engine; it reads Scheme programs, compiles it on-the-fly and executes it on a virtual machine. Sagittarius **Mostly** conforms the language standard "Revised^6 Report on the Algorithmic Language Scheme" (R6RS), and supports various common libraries defined in SRFIs

There are a lot of Scheme implementations and it has different strong and weak point. Frankly speaking, Sagittarius does not have much competitive point from other implementation. The only thing it has “Flexibility”. R6RS specifies strict requirements, but sometimes you may think this is too much. For that purpose, Sagittarius has less strictness but it makes not to conform the requirements. That's why it's Mostly R6RS.

To avoid to lose portability or write miss-working code, you may want to know what are the non conformed points.

Reader

Reader has two mode. One is R6RS mode and other one is compatible mode. Default mode is R6RS you can switch with *#!compatible* hash-bang which describe later. In compatible mode, reader also can read keyword which start with “:” and more non R6RS feature will be available.

Macro expansion

On R6RS requires explicit macro expansion phase, however in Sagittarius we do not have it. A macro will expand when programs will be compiled.

Unbound symbol

If you write unbound symbol in your code, Sagittarius, however, won't raise error until it really called. R6RS does not allow this behaviour. And also export symbol. If you do not define exported symbol, Sagittarius, then, won't raise error until it will be called. I'm not sure if this is convenient or not. So this behaviour may be changed.

Toplevel

Sagittarius does not require toplevel expression which is specified in R6RS.

Miscellaneous

Redefinition of exported values are allowed. The value which imported at the last will be used.

Almost everything is first class object. The syntax keyword “*export*” is not.

1.2 Notations

In this manual, each entry is represented like this.

`Category:` **foo** *arg1 arg2*

[spec] Description foo …

Category denotes category of the entry **foo**. The following category will appear in this manual.

Function	A Scheme function
Syntax	A syntax
Macro	A macro
Library	A library

For functions, syntaxes, or macros, the the entry may be followed by one or more arguments. In the argumuents list, following notations may appear.

arg ...
Indicates zero or more arguments

:optional x y z

:optional (x x-default) (y y-default) (z z-default)

Indicates is may take up to three optional arguments. The second form specifies default values for *x* and *y*. This is either builtin functions or closures which defined with *define-optional* macro.

The description of the entry follows the entry line. If the specification of the entry comes from some standard or implementation, its origin is noted in the bracket at the beginning of the description. The following origins are noted:

[R6RS]

[R6RS+]

The entry works as specified in “Revised^6 Report of Algorithmic Language Scheme.”. If it is marked as "[R6RS+]", the entry has additional functionality.

[SRFI-*n*]

[SRFI-*n*+]

The entry works as specified in SRFI-*n*. If it is marked as "[SRFI-*n*+]", the entry has additional functionality.

2 Programming in Sagittarius

2.1 Invoking sash

Sagittarius can be used either as an independent Schame interpreter or an embedded Scheme librar. The interpreter which comes with Sagittarius destribution is a program named *sash*.

Program:**sash** *[options] scheme-file arg ...*

Unfortunately, *sash* does not have interactive mode for now. So you always need to give a *scheme-file* to *sash*. Detail *options* are given with option *"-h"*.

2.2 Writing Scheme scripts

When a Scheme file is given to *sash*, it bounds an internal variable to list of the remaining command-line arguments which you can get with *command-line* procedure, then loads the Scheme program. If the first line of *scheme-file* begins with "*#!*", then *sash* ignores the entire line. This is useful to write a Scheme program that works as an executable script in unix-like systems.

Typical Sagittarius script has the first line like this:

```
#!/usr/local/bin/sash
```

or,

```
#!/bin/env sash
```

The second form uses "*shell trampoline*" technique so that the script works as far as *sash* is in the PATH.

After the script file is successfully loaded, then *sash* will process all toplevel expression the same as Perl.

Now I show a simple example below. This script works like *cat(1)*, without any command-line option processing and error handling.

```
#!/usr/local/bin/sash
(import (rnrs))
(let ((args (command-line)))
  (unless (null? (cdr args))
    (for-each (lambda (file)
          (call-with-input-file file
            (lambda (in)
              (display (get-string-all in)))))
          (cdr args)))
  0)
```

3 R6RS Libraries

Sagittarius works with library even toplevel expressions are belong to a library named "*user*". Here I list up all R6RS libraries. Some libraries contain the same procedure ie. *assoc* which is in *(rnrs (6))* and *(srfi :1 lists)*. In this case I will put a pointer to other library's section.

If library specifies its version, Sagittarius, however, ignores it. This behaviour may change in future.

3.1 (rnrs (6))

Library: **(rnrs (6))**

[R6RS] The library *(rnrs (6)* is required by R6RS. It just export all symbols from the libraries which are listed below.

Most of these libraries documentations are from R6RS specification.

- (rnrs base (6))
- (rnrs unicode (6)
- (rnrs bytevector (6))
- (rnrs lists (6))
- (rnrs sorting (6))
- (rnrs control (6))
- (rnrs records syntactic (6))
- (rnrs records procedural (6))
- (rnrs records inspection (6))
- (rnrs exceptions (6))
- (rnrs conditions (6))
- (rnrs io ports (6))
- (rnrs io simple (6))
- (rnrs files (6))
- (rnrs programs (6))
- (rnrs arithmetic fixnums (6))
- (rnrs arithmetic flonums (6))
- (rnrs arithmetic bitwise (6))
- (rnrs syntax-case (6))
- (rnrs hashtables (6))
- (rnrs enums (6))

3.2 (rnrs base (6))

Library:**(rnrs base (6))**

[R6RS] This library exports many of the procedure and syntax bindings that are traditionally associated with Scheme.

3.2.1 Variable definitions

Syntax:**define** *variable expression*
Syntax:**define** *variable*
Syntax:**define** *(variable formals) body ...*
Syntax:**define** *(variable . formal) body ...*

[R6RS] The *define* form is a *definition* used to create variable bindings and may appear anywhere other definitions may appear.

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

```
(define add3 (lambda (x) (+ x 3)))

(add3 3)                    ⇒ 6

(define first car)

(first '(1 2))              ⇒ 1
```

The second form of define is equivalent to

(define *variable unspecified*)

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

In the third form of define, *formals* must be either a sequence of zero or more variables, or a sequence of one or more variables followed by a dot . and another variable. This form is equivalent to

(define *variable* (lambda (*formals*) *body ...*)).

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

(define *variable* (lambda *formal body ...*)).

3.2.2 Syntax definitions

Syntax:**define-syntax** *keyword expression*

[R6RS] The define-syntax form is a *definition* used to create keyword bindings and may appear anywhere other definitions may appear.

Binds *keyword* to the value of *expression*, which must evaluate, at macro-expansion time, to a transformer.

3.2.3 Quotation

Syntax:**quote** *datum*

[R6RS] *quote* evaluates to the datum value represented by *datum*

```
(quote a)               ⇒ a

(quote #(a b c))        ⇒ #(a b c)

(quote (+ 1 2))         ⇒ (+ 1 2)
```

3.2.4 Procedures

Syntax: **lambda** *formals body ...*

[R6RS]A *lambda* expression evaluates to a procedure. The environment in effect when the lambda expression is evaluated is remembered as part of the procedure. When the procedure is later called with some arguments, the environment in which the *lambda* expression was evaluated is extended by binding the variables in the parameter list to fresh locations, and the resulting argument values are stored in those locations. Then, the expressions in the *body* of the *lambda* expression are evaluated sequentially in the extended environment. The results of the last expression in the body are returned as the results of the procedure call.

```
(lambda (x) (+ x x))          ⇒ a procedure

((lambda (x) (+ x x)) 4)      ⇒ 8

((lambda (x)

  (define (p y) (+ y 1))

  (+ (p x) x)) 5)             ⇒11



(define reverse-subtract (lambda (x y) (- y x)))

(reverse-subtract 7 10)       ⇒ 3



(define add4 (let ((x 4)) (lambda (y) (+ x y))))

(add4 6)                      ⇒ 10
```

Formals must have one of the following forms:

- (<*variable1*> ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments are stored in the bindings of the corresponding variables.
- <*variable*>: The procedure takes any number of arguments; when the procedure is called, the sequence of arguments is converted into a newly allocated list, and the list is stored in the binding of the <*variable*>.
- (<*variable1*> ... <*variablen*> . <*variablen+1*>): If a period . precedes the last variable, then the procedure takes n or more arguments, where n is the number of parameters before the period (there must be at least one). The value stored in the binding of the last variable is a newly allocated list of the arguments left over after all the other arguments have been matched up against the other parameters.

  ```
  ((lambda x x) 3 4 5 6)          ⇒ (3 4 5 6)

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

Any *variable* must not appear more than once in *formals*.

3.2.5 Conditionals

Syntax: **if** *test consequent alternate*

Syntax: **if** *test consequent*

[R6RS]An *if* expression is evaluated as follows: first, *test* is evaluated. If it yields a true value, then *consequent* is evaluated and its values are returned. Otherwise *alternate* is evaluated and its values are returned. If *test* yields #f and no *alternate* is specified, then the result of the expression is unspecified.

3.2.6 Assignment

Syntax: **set!** *variable expression*

[R6RS] *Expression* is evaluated, and the resulting value is stored in the location to which *variable* is bound. *Variable* must be bound either in some regionenclosing the set! expression or at the top level. The result of the set! expression is unspecified.

NB: R6RS requires to throw syntax violation if *variable* referes immutable binding. In Sagittarius, however, it won't throw any error.

3.2.7 Derived conditionals

Syntax: **cond** *clause ...*

[R6RS] Each *clause* must be the form

(test expression ...)

(test => expression)

*(*else *expression ...)*

The last form can appear only in the last clause.

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

Syntax: **case** *key clause ...*

[R6RS] *Key* must be an expression. Each *clause* must have one of the following forms:

((*datum ...*) *expression ...*)

(else *expression …*)

The last form can appear only in the last clause.

A *case* expression is evaluated as follows. *Key* is evaluated and its result is compared using eqv? against the data represented by the *datum*s of each *clause* in turn, proceeding in order from left to right through the set of clauses. If the result of evaluating *key* is equivalent to a datum of a *clause*, the corresponding *expression*s are evaluated from left to right and the results of the last expression in the *clause* are returned as the results of the case expression. Otherwise, the comparison process continues. If the result of evaluating *key* is different from every datum in each set, then if there is an else clause its expressions are evaluated and the results of the last are the results of the case

expression; otherwise the case expression returns unspecified values.

Syntax: **and** *test* ...

[R6RS] If there are no *test*s, #t is returned. Otherwise, the *test* expressions are evaluated from left to right until a *test* returns #f or the last *test* is reached. In the former case, the and expression returns #f without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

(and (= 2 2) (> 2 1)) ⇒ #t

(and (= 2 2) (< 2 1)) ⇒ #f

(and 1 2 'c '(f g)) ⇒ (f g)

(and) ⇒ #t

Syntax: **or** *test* ...

[R6RS] If there are no *test*s, #f is returned. Otherwise, the *test* expressions are evaluated from left to right until a *test* returns a true value or the last *test* is reached. In the former case, the or expression returns val without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

(or (= 2 2) (> 2 1)) ⇒ #t

(or (= 2 2) (< 2 1)) ⇒ #t

(or #f #f #f) ⇒ #f

(or '(b c) (/ 3 0)) ⇒ (b c)

3.2.8 Binding constructs

Syntax: **let** *bindings body* ...

[R6RS] *Bindings* must have the form

((*variable1 init1*) ...),

where each *init* is an expression.Any variable must not appear more than once in the *variable*s.

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

Syntax: **let*** *bindings body* ...

[R6RS] *Bindings* must have the form

((*variable1 init1*) ...),

where each *init* is an expression.

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

Syntax: **letrec** *bindings body ...*

[R6RS]*Bindings* must have the form

((*variable1 init1*) ...),

where each *init* is an expression.Any variable must not appear more than once in the *variable*s.

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

It should be possible to evaluate each *init* without assigning or referring to the value of any *variable*. In the most common uses of letrec, all the *init*s are *lambda* expressions and the restriction is satisfied automatically.

Syntax: **letrec*** *bindings body ...*

[R6RS]*Bindings* must have the form

((*variable1 init1*) ...),

where each *init* is an expression.Any variable must not appear more than once in the *variable*s.

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

It must be possible to evaluate each *init* without assigning or referring to the value of the corresponding *variable* or the *variable* of any of the bindings that follow it in *bindings*.

Syntax: **let-values** *mv-bindings body ...*

[R6RS]*Mv-vindings* must have the form

((formals init1) ...),

where each *init* is an expression. Any variable must not appear more than once in the set of *formals*.

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

Syntax: **let-values*** *mv-bindings body ...*

[R6RS]*Mv-vindings* must have the form

((formals init1) ...),

where each *init* is an expression. In each *formals*, any variable must not appear more than once.

The *let*-values* form is similar to *let-values*, but the *init*s are evaluated and bindings created sequentially from left to right, with the regionof the bindings of each *formals* including the bindings to its right as well as *body*. Thus the second *init* is evaluated in an environment in which the

bindings of the first *formals* is visible and initialized, and so on.

3.2.9 Sequencing

Syntax:**begin** *form ...*
Syntax:**begin** *expression ...*

[R6RS]The *begin* keyword has two different roles, depending on its context:

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

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

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

3.2.10 Equivalence predicates

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

Function:**eq?** *obj1 obj2*
Function:**eqv?** *obj1 obj2*
Function:**equal?** *obj1 obj2*

[R6RS] *eq?* only sees if the given two objects are the same object or not, *eqv? co*mpares numbers.*equal?* compares the values equivalence.

On Sagittarius Scheme interned symbol, keyword(only compatible mode), character, literal string, boolean, fixnum, and '() are used as the same objects. If these objects indicates the same value then eq? Returns #t.

The following examples are not specified R6RS. But it is always good to know how it works.

```
(let ((p (lambda (x) x)))

  (eqv? p p))              ⇒#t

(eqv? "" "")               ⇒#t (literal strings are the same objects, so even “abc” and “abc” return #t)

(eqv? '#() '#())           ⇒#f

(eqv? (lambda (x) x)

      (lambda (x) x))      ⇒#f

(eqv? (lambda (x) x)
```

(lambda (y) y) ⇒#f

(eqv? +nan.0 +nan.0) ⇒#f

3.2.11 Procedure predicate

Function:**procedure?** *obj*

[R6RS] Returns #t if obj is a procedure, otherwise returns #f.

3.2.12 Numerical type predicates

Function:**number?** *obj*
Function:**complex?** *obj*
Function:**real?** *obj*
Function:**rational?** *obj*
Function:**integer?** *obj*

[R6RS] These numerical type predicates can be applied to any kind of argument. They return #t if the object is a number object of the named type, and #f otherwise. In general, if a type predicate is true of a number object then all higher type predicates are also true of that number object. Consequently, if a type predicate is false of a number object, then all lower type predicates are also false of that number object.

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

If *x* is a real number object, then (*rational? x*) is true if and only if there exist exact integer objects *k1* and *k2* such that (= *x* (*/ k1 k2*)) and (= (*numerator x*) *k1*) and (= (*denominator x*) *k2*) are all true. Thus infinities and NaNs are not rational number objects.

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

Function:**real-valued?** *obj*
Function:**rational-valued?** *obj*
Function:**integer-valued?** *obj*

[R6RS] These numerical type predicates can be applied to any kind of argument. The *real-valued?* procedure returns #t if the object is a number object and is equal in the sense of = to some real number object, or if the object is a NaN, or a complex number object whose real part is a NaN and whose imaginary part is zero in the sense of *zero?*. The *rational-valued?* and *integer-valued?* procedures return #t if the object is a number object and is equal in the sense of = to some object of the named type, and otherwise they return #f.

Function:**exact?** *obj*
Function:**inexact?** *obj*

[R6RS] These numerical predicates provide tests for the exactness of a quantity. For any number object, precisely one of these predicates is true.

3.2.13 Generic conversion

Function:**exact** *z*
Function:**inexact** *z*

[R6RS] The *inexact* procedure returns an inexact representation of *z*. If inexact number objects of the appropriate type have bounded precision, then the value returned is an inexact number object that is nearest to the argument.

The *exact* procedure returns an exact representation of *z*. The value returned is the exact number object that is numerically closest to the argument; in most cases, the result of this procedure should be numerically equal to its argument.

3.2.14 Arithmetic operations

Function:= *z1 z2 z3 ...*
Function:> *x1 x2 x3 ...*
Function:< *x1 x2 x3 ...*
Function:>= *x1 x2 x3 ...*
Function:<= *x1 x2 x3 ...*

[R6RS] These procedures return #t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, and #f otherwise.

Function:**zero?** *z*
Function:**positive?** *z*
Function:**negative?** *z*
Function:**odd?** *z*
Function:**even?** *z*
Function:**finite?** *z*
Function:**infinite?** *z*
Function:**nan?** *z*

[R6RS] These numerical predicates test a number object for a particular property, returning #t or #f. The *zero?* procedure tests if the number object is = to zero, *positive?* tests whether it is greater than zero, *negative?* tests whether it is less than zero, *odd?* tests whether it is odd, *even?* tests whether it is even, *finite?* tests whether it is not an infinity and not a NaN, *infinite?* tests whether it is an infinity, *nan?* tests whether it is a NaN.

Function:**max** *x1 x2 ...*
Function:**min** *x1 x2 ...*

[R6RS] These procedures return the maximum or minimum of their arguments.

Function:+ *z ...*
Function:* *z ...*

[R6RS] These procedures return the sum or product of their arguments.

Function:- *z ...*
Function:- *z1 z2 ...*

[R6RS] With two or more arguments, this procedures returns the difference of its arguments, associating to the left. With one argument, however, it returns the additive inverse of its argument.

If this procedure is applied to mixed non-rational real and non-real complex arguments, it returns an unspecified number object.

Function:/ *z ...*
Function:/ *z1 z2 ...*

[R6RS+] If all of the arguments are exact, then the divisors must all be nonzero. With two or more arguments, this procedure returns the quotient of its arguments, associating to the left. With one argument, however, it returns the multiplicative inverse of its argument.

If this procedure is applied to mixed non-rational real and non-real complex arguments, it returns an unspecified number object.

In R6RS, it requires to raise *&assertion* when the divisor was 0, on Sagittarius, however, it

returns NaN or infinite number when it is running with compatible mode. In R6RS mode it raises *&assertion*.

Function: **abs** *x*

[R6RS] Returns the absolute value of its argument.

Function: **div-and-mod** *x1 x2*
Function: **div** *x1 x2*
Function: **mod** *x1 x2*
Function: **div0-and-mod0** *x1 x2*
Function: **div0** *x1 x2*
Function: **mod0** *x1 x2*

[R6RS] These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations. In each case, *x1* must be neither infinite nor a NaN, and *x2* must be nonzero; otherwise, an exception with condition type *&assertion* is raised.

Function: **gcd** *n1 ...*
Function: **lcm** *n1 ...*

[R6RS] These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative.

Function: **numerator** *q*
Function: **denominator** *q*

[R6RS] These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

Function: **floor** *x*
Function: **ceiling** *x*
Function: **truncate** *x*
Function: **round** *x*

[R6RS] These procedures return inexact integer objects for inexact arguments that are not infinities or NaNs, and exact integer objects for exact rational arguments. For such arguments, *floor* returns the largest integer object not larger than *x*. The *ceiling* procedure returns the smallest integer object not smaller than *x*. The *truncate* procedure returns the integer object closest to *x* whose absolute value is not larger than the absolute value of *x*. The *round* procedure returns the closest integer object to *x*, rounding to even when *x* represents a number halfway between two integers.

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

Function: **rationalize** *x1 x2*

[R6RS] The *rationalize* procedure returns the a number object representing the *simplest* rational number differing from *x1* by no more than *x2*. A rational number r1 is simpler than another rational number r2 if r1 = p1/q1 and r2 = p2/q2 (in lowest terms) and $|p1| \leq |p2|$ and $|q1| \leq |q2|$. Thus 3/5 is simpler than 4/7. Although not all rationals are comparable in this ordering (consider 2/7 and 3/5) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler 2/5 lies between 2/7 and 3/5). Note that 0 = 0/1 is the simplest rational of all.

Function: **exp** *x*
Function: **log** *x*
Function: **log** *x1 x2*
Function: **sin** *x*
Function: **cos** *x*
Function: **tan** *x*

Function:**asin** *x*
Function:**acos** *x*
Function:**atan** *x*
Function:**atan** *x1 x2*

[R6RS] These procedures compute the usual transcendental functions. The *exp* procedure computes the base-e exponential of *z*. The *log* procedure with a single argument computes the natural logarithm of *z* (not the base-ten logarithm); (*log z1 z2*) computes the base-*z2* logarithm of *z1*. The *asin*, *acos*, and *atan* procedures compute arcsine, arccosine, and arctangent, respectively. The two-argument variant of *atan* computes (*angle* (*make-rectangular x2 x1*)).

Function:**sqrt** *z*

[R6RS] Returns the principal square root of z. For rational z, the result has either positive real part, or zero real part and non-negative imaginary part.

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

Function:**expt** *z1 z2*

[R6RS] Returns *z1* raised to the power *z2*. For nonzero *z1*, this is $e^{x2 \log z1}$. 0.0z is 1.0 if *z* = 0.0, and 0.0 if (r*eal-part z*) is positive. For other cases in which the first argument is zero, either an exception is raised with condition type *&implementation-restriction*, or an unspecified number object is returned.

For an exact real number object *z1* and an exact integer object *z2*, (*expt z1 z2*) must return an exact result. For all other values of z1 and *z2*, (*expt z1 z2*) may return an inexact result, even when both *z1* and *z2* are exact.

Function:**make-rectangular** *x1 x2*
Function:**make-polar** *x3 x4*
Function:**real-part** *z*
Function:**imag-part** *z*
Function:**magnitude** *z*
Function:**angle** *z*

[R6RS] Suppose *a1, a2, a3*, and *a4* are real numbers, and *c* is a complex number such that the following holds:

$$c = a1 + a2^{i} = a3 e^{ia4}$$

Then, if *x1, x2, x3*, and *x4* are number objects representing *a1, a2, a3*, and *a4*, respectively, (*make-rectangular x1 x2*) returns *c*, and (*make-polar x3 x4*) returns *c*.

3.2.15 Numerical input and output

Function:**number->string** *z :optional radix precision*

[R6RS+] *Radix* must be an exact integer object, between 2, and 32. If omitted, radix defaults to 10. In Sagittarius *precision* will be ignored. The *number->string* procedure takes a number object and a radix and returns as a string an external representation of the given number object in the given radix such that

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

is true.

Function:**string->number** *string :optional radix*

[R6RS+] Returns a number object with maximally precise representation expressed by the given string. *Radix* must be an exact integer object, between 2, and 32. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g., "#o177"). If *radix* is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number object or a notation for a rational number object with a zero denominator, then *string->number* returns #f.

These *number->string* and *string->number*'s resolutions of *radix* are taken from Gauche.

3.2.16 Booleans

Function:**not** *obj*

[R6RS] Returns #t if *obj* is #f, and returns #f otherwise.

Function:**boolean?** *obj*

[R6RS] Returns #t if *obj* is either #t or #f and returns #f otherwise.

Function:**boolean=?** *bool1 bool2 ...*

[R6RS] Returns #t if the booleans are the same.

3.2.17 Pairs and lists

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

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

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

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

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

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

(a b c . d)

is equivalent to

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

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

Function:**pair?** *obj*

[R6RS] Returns #t if obj is a pair, and otherwise returns #f.

Function:**cons** *obj1 obj2*

[R6RS] Returns a newly allocated pair whose car is *obj1* and whose cdr is *obj2*. The pair is guaranteed to be different (in the sense of eqv?) from every existing object.

Function:**car** *pair*

Function:**cdr** *pair*

[R6RS] Returns the contents of the *car/cdr* field of *pair*.

Function:**caar** *pair*
Function:**cadr** *pair*
:
Function:**cadddr** *pair*
Function:**cddddr** *pair*

[R6RS] These procedures are compositions of car and cdr, where for example *caddr* could be defined by

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

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

Function:**null?** *obj*

[R6RS] Returns #t if obj is the empty list, #fotherwise.

Function:**list?** *obj*

[R6RS] Returns #t if obj is a list, #f otherwise. By definition, all lists are chains of pairs that have finite length and are terminated by the empty list.

Function:**length** *list*

[R6RS+] Returns the length of *list*. If the *list* is *improper list*, it returns -1 If the *list* is *infinite list* , it returns -2.

Function:**append** *list ... obj*

[R6RS] Returns a possibly improper list consisting of the elements of the first *list* followed by the elements of the other *lists*, with *obj* as the cdr of the final pair. An improper list results if *obj* is not a list.

Function:**reverse** *list*

[R6RS] Returns a newly allocated list consisting of the elements of *list* in reverse order.

Function:**list-tail** *list k :optional fallback*

[R6RS+] The *list-tail* procedure returns the subchain of pairs of *list* obtained by omitting the first *k* elements.If *fallback* is given and *k* is out of range, it returns *fallback* otherwise throw *error*.

Function:**list-ref** *list k :optional fallback*

[R6RS+] The *list-ref* procedure returns the *k*th element of list.If *fallback* is given and *k* is out of range, it returns *fallback* otherwise throw *error*.

Note: The error thrown by the *list-tail* or *list-ref*is just *string*. This may be replaced by *&assertion*.

These *list-tail* and *list-ref*'s behaviours are taken from Gauche.

Function:**map** *proc list1 list2*...

[R6RS+] The *map* procedure applies *proc* element-wise to the elements of the *lists* and returns a list of the results, in order. The order in which *proc* is applied to the elements of the lists is unspecified. If multiple returns occur from *map*, the *values* returned by earlier returns are not mutated. If the given *lists* are not the same length, when the shortest list is processed the *map* will stop.

Function:**for-each** *proc list1 list2 ...*

[R6RS+] The *for-each* procedure applies *proc* element-wise to the elements of the *lists* for its side effects, in order from the first elements to the last. The return values of for-each are unspecified. If the given *lists* are not the same length, when the shortest list is processed the *for-each* will stop.

These *map* and *for-each*'s behaviours are taken from Gauche.

3.2.18 Symbols

Function:**symbol?** *obj*
[R6RS] Returns #t if obj is a symbol, otherwise returns #f.

Function:**symbol->string** *symbol*
[R6RS] Returns the name of *symbol* as an *immutable string*.

Function:**symbol=?** *symbol1 symbol2 ...*
[R6RS] Returns #t if the *symbols* are the same, i.e., if their names are spelled the same.

Function:**string->symbol** *string*
[R6RS] Returns the symbol whose name is *string*.

3.2.19 Characters

Characters are objects that represent Unicode scalar values.

Function:**char?** *obj*
[R6RS] Returns #t if obj is a character, otherwise returns #f.

Function:**char->integer** *char*
Function:**integer->char** *sv*
[R6RS] *Sv* must be a Unicode scalar value, i.e., a non-negative exact integer object in [0, #xD7FF] ∪ [#xE000, #x10FFFF].

Given a character, *char->integer* returns its Unicode scalar value as an exact integer object. For a Unicode scalar value *sv*, *integer->char* returns its associated character.

Function:**char=?** *char1 char2 char3 ...*
Function:**char=?** *char1 char2 char3 ...*
Function:**char=?** *char1 char2 char3 ...*
Function:**char=?** *char1 char2 char3 ...*
[R6RS] These procedures impose a total ordering on the set of characters according to their Unicode scalar values.

3.2.20 Strings

Strings are sequences of characters.

The *length* of a string is the number of characters that it contains. This number is fixed when the string is created. The *valid indices* of a string are the integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

Function:**string?** *obj*
[R6RS] Returns #t if obj is a string, otherwise returns #f.

Function:**make-string?** *k :optional char*
[R6RS] Returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the string are *#\space*.

These are equivalence:

(make-string 10) = (make-string 10 #\space)

Function:**string** *char ...*
[R6RS] Returns a newly allocated string composed of the arguments.

Function: **string-length** *string*

[R6RS] Returns the number of characters in the given *string* as an exact integer object.

Function: **string-ref** *string k :optional fallback*

[R6RS+] The string-ref procedure returns character. If a third argument is given ant *k* is not a valid index, it returns *fallback*, otherwise raises *&assertion*.

k of string using zero-origin indexing.

Function: **string=?** *string1 string2 string3 ...*

[R6RS] Returns #t if the *strings* are the same length and contain the same characters in the same positions. Otherwise, the *string=?* procedure returns #f.

Function: **string<?** *string1 string2 string3 ...*
Function: **string>?** *string1 string2 string3 ...*
Function: **string<=?** *string1 string2 string3 ...*
Function: **string>=?** *string1 string2 string3 ...*

[R6RS] These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, *string<?* is the lexicographic ordering on strings induced by the ordering *char<?* on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

Function: **substring** *string start end*

[R6RS] *String* must be a string, and *start* and *end* must be exact integer objects satisfying

0 ≤ start ≤ end ≤ (string-length string).

The *substring* procedure returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

Function: **string-append** *string ...*

[R6RS] Returns a newly allocated string whose characters form the concatenation of the given strings.

Function: **string->list** *string*
Function: **list->string** *list*

[R6RS] List must be a list of characters. The *string->list* procedure returns a newly allocated list of the characters that make up the given *string*. The *list->string* procedure returns a newly allocated string formed from the characters in *list*. The *string->list* and *list->string* procedures are inverses so far as *equal?* is concerned.

Function: **string-for-each** *proc string1 string2 ...*

[R6RS+] *Proc* should accept as many arguments as there are *strings*. The *string-for-each* procedure applies *proc* element-wise to the characters of the *strings* for its side effects, in order from the first characters to the last. The return values of *string-for-each* are unspecified.

Analogous to *for-each*.

Function: **string-copy** *string*

[R6RS] Returns a newly allocated copy of the given *string*.

3.2.21 Vectors

Vectors are heterogeneous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time needed to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indices*of a vector are the exact non-

negative integer objects less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

In R6RS *vector constants* must be quoted like list costants, on Sagittarius, however, it does not have to be. So you can write it either way.

Function: **vector?** *obj*

[R6RS] Returns #t if *obj* is a vector. Otherwise the procedure returns #f.

Function: **make-vector** *k :optional fill*

[R6RS] Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

Function: **vector** *obj ...*

[R6RS] Returns a newly allocated vector whose elements contain the given arguments. Analogous to list.

Function: **vector-length** *vector*

[R6RS] Returns the number of elements in *vector* as an exact integer object.

Function: **vector-ref** *vector k :optional fallback*

[R6RS+] The *vector-ref* procedure returns the contents of element *k* of *vector*. If a third argument is given and *k* is not a valid index, it returns *fallback,* otherwise raises *&assertion*.

Function: **vector-set!***vector k obj*

[R6RS] *K* must be a valid index of *vector*. The *vector-set!* procedure stores *obj* in element *k* of *vector*, and returns unspecified values.

It raises *&assertion* when it attempt to modify immutable vector on R6RS mode.

Function: **vector->list** *vector :optional start end*
Function: **list->vector** *list :optional start end*

[R6RS+] The *vector->list* procedure returns a newly allocated list of the objects contained in the elements of *vector*. The *list->vector* procedure returns a newly created vector initialized to the elements of the list *list*. The optional *start* and *end* arguments limit the range of the source.

```
(vector->list '#(1 2 3 4 5))     ⇒(1 2 3 4 5)

(list->vector '(1 2 3 4 5))      ⇒#(1 2 3 4 5)

(vector->list '#(1 2 3 4 5) 2 4) ⇒(3 4)

(list->vector (circular-list 'a 'b 'c) 1 6)  ⇒#(b c a b c)
```

Function: **vector-fill!***vector fill :optional start end*

[R6RS+] Stores *fill* in every element of *vector* and returns unspecified values. Optional *start* and *end* limits the range of effect between *start*-th index (inclusive) to *end*-th index (exclusive). *Start* defaults to zero, and *end* defaults to the length of vector.

Function: **vector-map** *proc vector1 vector2 ...*

[R6RS+] *Proc* should accept as many arguments as there are *vectors*. The *vector-map* procedure applies *proc* element-wise to the elements of the *vectors* and returns a vector of the results, in order. If multiple returns occur from *vector-map*, the return values returned by earlier returns are not mutated.

Analogous to *map*.

Function:**vector-for-each** *proc vector1 vector2 ...*

[R6RS+] *Proc* should accept as many arguments as there are *vectors*. The *vector-for-each* procedure applies *proc* element-wise to the elements of the *vectors* for its side effects, in order from the first elements to the last. The return values of *vector-for-each* are unspecified.

Analogous to *for-each*.

3.2.22 Errors and violations

Function:**error** *who message irritant ...*

Function:**assertion-violation** *who message irritant ...*

[R6RS] *Who* must be a string or a symbol or #f. *Message* must be a string. The *irritants* are arbitrary objects.

These procedures raise an exception. The *error* procedure should be called when an error has occurred, typically caused by something that has gone wrong in the interaction of the program with the external world or the user. The *assertion-violation* procedure should be called when an invalid call to a procedure was made, either passing an invalid number of arguments, or passing an argument that it is not specified to handle.

The *who* argument should describe the procedure or operation that detected the exception. The *message* argument should describe the exceptional situation. The *irritants* should be the arguments to the operation that detected the operation.

3.2.23 Control features

Function:**apply** *proc arg1 ...rest-args*

[R6RS] *Rest-args* must be a list. *Proc* should accept *n* arguments, where n is number of *args* plus the length of *rest-args*. The *apply* procedure calls *proc* with the elements of the list *(append (list arg1 ...) rest-args)* as the actual arguments.

If a call to *apply* occurs in a tail context, the call to *proc* is also in a tail context.

Function:**call-with-current-continuation** *proc*

Function:**call/cc** *proc*

[R6RS] *Proc* should accept one argument. The procedure *call-with-current-continuation* (which is the same as the procedure *call/cc*) packages the current continuation as an “escape procedure”and passes it as an argument to *proc*. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead reinstate the continuation that was in effect when the escape procedure was created. Calling the escape procedure may cause the invocation of *before* and *after* procedures installed using *dynamic-wind*.

The escape procedure accepts the same number of arguments as the continuation of the original call to *call-with-current-continuation*.

Function:**values** *obj ...*

[R6RS] Delivers all of its arguments to its continuation. The values procedure might be defined as follows:

Function:**call-with-values** *producer consumer*

[R6RS] *Producer* must be a procedure and should accept zero arguments. *Consumer* must be a procedure and should accept as many values as *producer* returns. The *call-with-values* procedure calls *producer* with no arguments and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to *call-with-values.*

If a call to *call-with-values* occurs in a tail context, the call to *consumer* is also in a tail context.

Function: **dynamic-wind** *before thunk after*

[R6RS] *Before*, *thunk*, and *after* must be procedures, and each should accept zero arguments. These procedures may return any number of values. The *dynamic-wind* procedure calls *thunk* without arguments, returning the results of this call. Moreover, *dynamic-wind* calls *before* without arguments whenever the dynamic extent of the call to *thunk* is entered, and *after* without arguments whenever the dynamic extent of the call to thunk is exited. Thus, in the absence of calls to escape procedures created by *call-with-current-continuation*, *dynamic-wind* calls *before*, *thunk*, and *after*, in that order.

3.2.24 Iteration

Syntax: **let** *variable bindings body ...*

[R6RS] "Named let" is a variant on the syntax of let that provides a general looping construct and may also be used to express recursion. It has the same syntax and semantics as ordinary let except that *variable* is bound within *body* to a procedure whose parameters are the bound variables and whose body is *body*. Thus the execution of *body* may be repeated by invoking the procedure named by *variable*.

3.2.25 Quasiquote

Syntax: **quasiquote** *qq-template*

Auxiliary syntax: **unquote** *qq-template*

Auxiliary syntax: **unquote-splicing** *qq-template*

[R6RS] "Quasiquote"expressions is useful for constructing a list or vector structure when some but not all of the desired structure is known in advance.

Qq-template should be as specified by the grammar at the end of this entry.

If no *unquote* or *unquote-splicing* forms appear within the *qq-template*, the result of evaluating (*quasiquote qq-template*) is equivalent to the result of evaluating (*quote qq-template*).

If an (*unquote expression ...*) form appears inside a *qq-template*, however, the *expression*s are evaluated ("*unquoted*") and their results are inserted into the structure instead of the *unquote* form.

If an (*unquote-splicing expression ...*) form appears inside a *qq-template*, then the *expression*s must evaluate to lists; the opening and closing parentheses of the lists are then "*stripped away*" and the elements of the lists are inserted in place of the *unquote-splicing* form.

Any *unquote-splicing* or multi-operand *unquote* form must appear only within a list or vector *qq-template*.

Note: even *unquote* and *unquote-splicing* are bounded, however it does not work with import prefix nor renamed import.

3.2.26 Binding constructs for syntactic keywords

The *let-syntax* and *letrec-syntax* forms bind keywords. On R6RS mode it works like a *begin* form, a *let-syntax* or *letrec-syntax* form may appear in a definition context, in which case it is treated as a definition, and the forms in the body must also be definitions. A *let-syntax* or *letrec-syntax* form may also appear in an expression context, in which case the forms within their bodies must be expressions.

Syntax: **let-syntax** *bindings form ...*

[R6RS] *Bindings* must have the form

((*keyword expression*) ...)

Each *keyword* is an identifier, and each *expression* is an expression that evaluates, at macro-expansion time, to a transformer. Transformers may be created by *syntax-rules* or *identifier-syntax* or by one of the other mechanisms described in library chapter on "*syntax-case*". It is a syntax violation for *keyword* to appear more than once in the list of keywords being bound.

The *form*s are expanded in the syntactic environment obtained by extending the syntactic environment of the *let-syntax* form with macros whose keywords are the *keyword*s, bound to the specified transformers. Each binding of a *keyword* has the *form*s as its region.

Syntax: **letrec-syntax** *bindings form ...*

[R6RS] *Bindings* must have the form

((*keyword expression*) ...)

Each *keyword* is an identifier, and each *expression* is an expression that evaluates, at macro-expansion time, to a transformer. Transformers may be created by *syntax-rules* or *identifier-syntax* or by one of the other mechanisms described in library chapter on "*syntax-case*". It is a syntax violation for *keyword* to appear more than once in the list of keywords being bound.

The *form*s are expanded in the syntactic environment obtained by extending the syntactic environment of the *letrec-syntax* form with macros whose keywords are the *keyword*s, bound to the specified transformers. Each binding of a *keyword* has the *bindings* as well as the <*orm*s within its region, so the transformers can transcribe forms into uses of the macros introduced by the *letrec-syntax* form.

Note: The *form*s of a *let-syntax* and a *letrec-syntax* form are treated, whether in definition or expression context, as if wrapped in an implicit *begin* on R6RS mode, it is, then, treated as if wrapped in an implicit *let* on compatible mode. Thus on compatible mode, it creates a scope.

3.2.27 Macro transformers

In R6RS, it requires '_'...' as bounded symbols but in Sagittarius these are not bound.And if import clause has *rename* or *prefix* these auxiliary syntax are not be renamed or prefixed. This behaivour may be fixed in future.

Syntax: **syntax-rules** *(literal ...) rule ...*

[R6RS] Each *literal* must be an identifier. Each *rule* must have the following form:

(sr*pattern template*)

An sr*pattern* is a restricted form of *pattern*, namely, a nonempty *pattern* in one of four parenthesized forms below whose first subform is an identifier or an underscore _. A *pattern* is an identifier, constant, or one of the following.

(*pattern ...*)

(*pattern pattern ... . pattern*)

(*pattern ... pattern ellipsis pattern ...*)

(*pattern ... pattern ellipsis pattern ... . pattern*)

#(*pattern ...*)

#(*pattern ... pattern ellipsis* pattern ...)

An *ellipsis* is the identifier "..." (three periods).

A *template* is a pattern variable, an identifier that is not a pattern variable, a pattern datum, or one of the following.

(*subtemplate ...*)

(*subtemplate ... . template*)

#(*subtemplate …*)

A *subtemplate* is a *template* followed by zero or more ellipses.

An instance of *syntax-rules* evaluates, at macro-expansion time, to a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by *syntax-rules* is matched against the patterns contained in the *rule*s, beginning with the leftmost *rule*. When a match is found, the macro use is transcribed hygienically according to the template. It is a *syntax violation* when no match is found.

Macro: **idetifier-syntax** *template*
Macro: **idetifier-syntax** *(id1 template1) ((set! id2 pattern) template2)*

[R6RS] The *id*s must be identifiers. The *template*s must be as for *syntax-rules*.

When a keyword is bound to a transformer produced by the first form of *identifier-syntax,* references to the keyword within the scope of the binding are replaced by *template*.

```
(define p (cons 4 5))
(define-syntax p.car (identifier-syntax (car p)))
p.car        ⇒4
(set! p.car 15)        ⇒ &syntax exception
```

The second, more general, form of identifier-syntax permits the transformer to determine what happens when *set!* is used. In this case, uses of the identifier by itself are replaced by *template1*, and uses of *set!* with the identifier are replaced by *template2*

```
(define p (cons 4 5))
(define-syntax p.car
  (identifier-syntax
    (_ (car p))
    ((set! _ e) (set-car! p e))))
(set! p.car 15)
p.car              ⇒15
p                  ⇒(15 5).
```

3.3 (rnrs unicode (6))

Library: **(rnrs unicode (6))**

[R6RS] The procedures exported by the (*rnrs unicode* (*6*))library provide access to some aspects of the Unicode semantics for characters and strings: category information, case-independent comparisons, case mappings, and normalization.

Some of the procedures that operate on characters or strings ignore the difference between upper case and lower case. These procedures have “-ci” (for “case insensitive”) embedded in their names.

3.3.1 Characters

Function: **char-upcase** *char*
Function: **char-downcase** *char*
Function: **char-titlecase** *char*
Function: **char-foldcase** *char*

[R6RS] These procedures take a character argument and return a character result. If the argument is an upper-case or title-case character, and if there is a single character that is its lower-case form, then *char-downcase* returns that character. If the argument is a lower-case or title-case character, and there is a single character that is its upper-case form, then *char-upcase* returns that character. If the argument is a lower-case or upper-case character, and there is a single character that is its title-case form, then *char-titlecase* returns that character. If the argument is not a title-case character and there is no single character that is its title-case form, then *char-titlecase* returns the upper-case form of the argument. Finally, if the character has a case-folded character, then *char-foldcase* returns that character. Otherwise the character returned is the same as the argument. For Turkic characters İ (#\x130) and ı (#\x131), char-foldcase behaves as the identity function; otherwise *char-foldcase* is the same as *char-downcase* composed with *char-upcase*.

Function: **char-ci=?** *char1 char2 char3 ...*
Function: **char-ci>?** *char1 char2 char3 ...*
Function: **char-ci<?** *char1 char2 char3 ...*
Function: **char-ci>=?** *char1 char2 char3 ...*
Function: **char-ci<=?** *char1 char2 char3 ...*

[R6RS] These procedures are similar to *char=?,* etc., but operate on the case-folded versions of the characters.

Function: **char-alphabetic?** *char*
Function: **char-numeric?** *char*
Function: **char-whitespace?** *char*
Function: **char-upper-case?** *char*
Function: **char-lower-case?** *char*
Function: **char-title-case?** *char*

[R6RS] These procedures return #t if their arguments are alphabetic, numeric, whitespace, upper-case, lower-case, or title-case characters, respectively; otherwise they return #f.

A character is alphabetic if it has the Unicode “Alphabetic” property. A character is numeric if it has the Unicode “Numeric” property. A character is whitespace if has the Unicode “White_Space” property. A character is upper case if it has the Unicode “Uppercase” property, lower case if it has the “Lowercase” property, and title case if it is in the Lt general category.

Function: **char-general-category** *char*

[R6RS] Returns a symbol representing the Unicode general category of char, one of Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Ps, Pe, Pi, Pf, Pd, Pc, Po, Sc, Sm, Sk, So, Zs, Zp, Zl, Cc, Cf, Cs, Co, or Cn.

3.3.2 Strings

Function: **string-upcase** *string*
Function: **string-downcase** *string*
Function: **string-titlecase** *string*
Function: **string-foldcase** *string*

[R6RS] These procedures take a string argument and return a string result. They are defined in terms of Unicode’s locale-independent case mappings from Unicode scalar-value sequences to scalar-value sequences. In particular, the length of the result string can be different from the length

of the input string. When the specified result is equal in the sense of *string=?* to the argument, these procedures may return the argument instead of a newly allocated string.

The *string-upcase* procedure converts a string to upper case; *string-downcase* converts a string to lower case. The *string-foldcase* procedure converts the string to its case-folded counterpart, using the full case-folding mapping, but without the special mappings for Turkic languages. The *string-titlecase* procedure converts the first cased character of each word via char-titlecase, and downcases all other cased characters.

Function:**string-ci=?** *string1 string2 string3 ...*
Function:**string-ci>?** *string1 string2 string3 ...*
Function:**string-ci<?** *string1 string2 string3 ...*
Function:**string-ci>=?** *string1 string2 string3 ...*
Function:**string-ci<=?** *string1 string2 string3 ...*

[R6RS] These procedures are similar to *string=?*, etc., but operate on the case-folded versions of the strings.

Function:**string-nfd** *string*
Function:**string-nfkd** *string*
Function:**string-nfc** *string*
Function:**string-nfkc** *string*

[R6RS] These procedures take a string argument and return a string result, which is the input string normalized to Unicode normalization form D, KD, C, or KC, respectively. When the specified result is equal in the sense of string=? to the argument, these procedures may return the argument instead of a newly allocated string.

3.4 (rnrs bytevectors (6)

Many applications deal with blocks of binary data by accessing them in various ways—extracting signed or unsigned numbers of various sizes. Therefore, the *(rnrs bytevectors (6))*library provides a single type for blocks of binary data with multiple ways to access that data. It deals with integers and floating-point representations in various sizes with specified endianness.

Bytevectorsare objects of a disjoint type. Conceptually, a bytevector represents a sequence of 8-bit bytes. The description of bytevectors uses the term byte for an exact integer object in the interval { -128, ..., 127} and the term octet for an exact integer object in the interval {0, ..., 255}. A byte corresponds to its two's complement representation as an octet.

The length of a bytevector is the number of bytes it contains. This number is fixed. A valid index into a bytevector is an exact, non-negative integer object less than the length of the bytevector. The first byte of a bytevector has index 0; the last byte has an index one less than the length of the bytevector.

Generally, the access procedures come in different flavors according to the size of the represented integer and the endianness of the representation. The procedures also distinguish signed and unsigned representations. The signed representations all use two's complement.

Like string literals, literals representing bytevectors do not need to be quoted:

#vu8(12 23 123) ⇒#vu8(12 23 123)

3.4.1 General operations

Macro:**endianness** *symbol*

[R6RS] The name of *symbol* must be a symbol describing an endianness. An implementation must support at least the symbols big and little, but may support other endianness symbols.

(*endianness symbol*) evaluates to the symbol named *symbol*. Whenever one of the procedures operating on bytevectors accepts an endianness as an argument, that argument must be one of these symbols. It is a syntax violation for *symbol* to be anything other than an endianness symbol supported by the implementation.

Function: **native-endianness**

[R6RS] Returns the endianness symbol associated implementation's preferred endianness (usually that of the underlying machine architecture). This may be any *symbol*, including a symbol other than big and little.

Function: **bytevector?** *obj*

[R6RS] Returns #t if obj is a bytevector, otherwise returns #f.

Function: **make-bytevector** *k :optional fill*

[R6RS] Returns a newly allocated bytevector of *k* bytes.

If the *fill* argument is missing, the initial contents of the returned bytevector are unspecified.

If the *fill* argument is present, it must be an exact integer object in the interval { - 128, ... 255} that specifies the initial value for the bytes of the bytevector: If *fill* is positive, it is interpreted as an octet; if it is negative, it is interpreted as a byte.

Function: **bytevector-length** *bytevector*

[R6RS] Returns, as an exact integer object, the number of bytes in *bytevector.*

Function: **bytevector=?** *bytevector1 bytevector2*

[R6RS] Returns #t if *bytevector1* and *bytevector2* are equal—that is, if they have the same length and equal bytes at all valid indices. It returns #f otherwise.

Function: **bytevector-fill!** *bytevector fill*

[R6RS] The *fill* argument is as in the description of the *make-bytevector* procedure. The *bytevector-fill!* procedure stores *fill* in every element of *bytevector* and returns unspecified values. Analogous to *vector-fill!*.

Function: **bytevector-copy!** *source source-start target target-start k*

[R6RS] *Source* and *target* must be bytevectors. *Source-start, target-start,* and *k* must be non-negative exact integer objects that satisfy

$$0 \le source - start \le source - start + k \le source - length$$
$$0 \le tartget - start \le target - start + k \le target - length$$

where *source-length* is the length of *source* and *target-length* is the length of *target*.

The *bytevector-copy!* procedure copies the bytes from source at indices

$$source - start\ , \ldots source - start + k - 1$$

to consecutive indices in *target* starting at *target-index*.

This must work even if the memory regions for the source and the target overlap, i.e., the bytes at the target location after the copy must be equal to the bytes at the source location before the copy.

This returns unspecified values.

Function: **bytevector-copy** *bytevector*

[R6RS] Returns a newly allocated copy of bytevector.

3.4.2 Operation on bytes and octets

Function: **bytevector-u8-ref** *bytevector k*
Function: **bytevector-s8-ref** *bytevector k*

[R6RS] *K* must be a valid index of *bytevector*.

The *bytevector-u8-ref* procedure returns the byte at index *k* of *bytevector*, as an octet.

The *bytevector-s8-ref* procedure returns the byte at index *k* of *bytevector*, as a (signed) byte.

```
(let ((b1 (make-bytevector 16 -127))
      (b2 (make-bytevector 16 255)))
  (list
    (bytevector-s8-ref b1 0)
    (bytevector-u8-ref b1 0)
    (bytevector-s8-ref b2 0)
    (bytevector-u8-ref b2 0)))
          ⇒(-127 129 -1 255)
```

Function: **bytevector-u8-set!** *bytevector k octet*
Function: **bytevector-s8-set!** *bytevector k byte*

[R6RS] *K* must be a valid index of *bytevector*.

The *bytevector-u8-set!* procedure stores *octet* in element *k* of *bytevector*.

The *bytevector-s8-set!* procedure stores the two's-complement representation of *byte* in element *k* of *bytevector*.

Both procedures return unspecified values.

```
(let ((b (make-bytevector 16 -127)))
  (bytevector-s8-set! b 0 -126)
  (bytevector-u8-set! b 1 246)
  (list
    (bytevector-s8-ref b 0)
    (bytevector-u8-ref b 0)
    (bytevector-s8-ref b 1)
    (bytevector-u8-ref b 1)))
          ⇒(-126 130 -10 246)
```

Function: **bytevector->u8-list** *bytevector*
Function: **u8-list->bytevector** *list*

[R6RS] *List* must be a list of octets.

The *bytevector->u8-list* procedure returns a newly allocated list of the octets of *bytevector* in the same order.

The *u8-list->bytevector* procedure returns a newly allocated bytevector whose elements are the elements of list *list*, in the same order. It is analogous to *list->vector*.

3.4.3 Operations on integers of arbitary size

Function: **bytevector-uint-ref** *bytevector k endianness size*
Function: **bytevector-sint-ref** *bytevector k endianness size*
Function: **bytevector-uint-set!** *bytevector k n endianness size*
Function: **bytevector-sint-set!** *bytevector k n endianness size*

[R6RS] *Size* must be a positive exact integer object. *K*, ..., *k + size - 1* must be valid indices of *bytevector*.

The *bytevector-uint-ref* procedure retrieves the exact integer object corresponding to the unsigned representation of size *size* and specified by *endianness* at indices *k, ..., k + size - 1*.

The *bytevector-sint-ref* procedure retrieves the exact integer object corresponding to the two's-complement representation of size *size* and specified by *endianness* at indices *k, ..., k + size - 1*.

For *bytevector-uint-set!*, *n* must be an exact integer object in the interval $0, \ldots, 256^{\text{size}} - 1$

The bytevector-uint-set! procedure stores the unsigned representation of size *size* and specified by *endianness* into *bytevector* at indices *k, ..., k + size - 1*.

For *bytevector-sint-set!*, *n* must be an exact integer object in the interval $-256^{\text{size}}/2, \ldots, 256^{\text{size}}/2 - 1$. *bytevector-sint-set!* stores the two's-complement representation of size *size* and specified by *endianness* into *bytevector* at indices *k, ..., k + size - 1*.

The ...*-set!* procedures return unspecified values.

Function: **bytevector->uint-list** *bytevector endianness size*
Function: **bytevector->sint-list** *bytevector endianness size*
Function: **uint-list->bytevector** *list endianness size*
Function: **sint-list->bytevector** *list endianness size*

[R6RS] *Size* must be a positive exact integer object. For *uint-list->bytevector*, *list* must be a list of exact integer objects in the interval $0, \ldots, 256^{\text{size}} - 1$. For *sint-list->bytevector*, *list* must be a list of exact integer objects in the interval $-256^{\text{size}}/2, \ldots, 256^{\text{size}}/2 - 1$. The length of *bytevector* or, respectively, of *list* must be divisible by *size*.

These procedures convert between lists of integer objects and their consecutive representations according to *size* and *endianness* in the bytevector objects in the same way as *bytevector->u8-list* and *u8-list->bytevector* do for one-byte representations.

3.4.4 Operation on 16-bit integers

Function: **bytevector-u16-ref** *bytevector k endianness*
Function: **bytevector-s16-ref** *bytevector k endianness*
Function: **bytevector-u16-native-ref** *bytevector k*
Function: **bytevector-s16-native-ref** *bytevector k*
Function: **bytevector-u16-set!** *bytevector k n endianness*
Function: **bytevector-s16-set!** *bytevector k n endianness*
Function: **bytevector-u16-native-set!** *bytevector k n*
Function: **bytevector-s16-native-set!** *bytevector k n*

[R6RS] *K* must be a valid index of *bytevector*; so must *k + 1*. For *bytevector-u16-set!* and *bytevector-u16-native-set!*, *n* must be an exact integer object in the interval $0, \ldots, 2^{16} - 1$. For *bytevector-s16-set!* and *bytevector-s16-native-set!*, *n* must be an exact integer object in the interval $-2^{15}, \ldots, 2^{15} - 1$.

These retrieve and set two-byte representations of numbers at indices *k* and *k + 1*, according to the endianness specified by *endianness*. The procedures with *u16* in their names deal with the unsigned representation; those with *s16* in their names deal with the two's-complement representation.

The procedures with native in their names employ the native endianness, and work only at aligned indices: *k* must be a multiple of 2.

The ...*-set!* procedures return unspecified values.

3.4.5 Operation on 32-bit integers

Function: **bytevector-u32-ref** *bytevector k endianness*
Function: **bytevector-s32-ref** *bytevector k endianness*
Function: **bytevector-u32-native-ref** *bytevector k*
Function: **bytevector-s32-native-ref** *bytevector k*
Function: **bytevector-u32-set!** *bytevector k n endianness*
Function: **bytevector-s32-set!** *bytevector k n endianness*
Function: **bytevector-u32-native-set!** *bytevector k n*
Function: **bytevector-s32-native-set!** *bytevector k n*

[R6RS] *K, ... k + 3* must be a valid indices of *bytevector*. For *bytevector-u32-set!* and *bytevector-u32-native-set!*, *n* must be an exact integer object in the interval $0, \ldots, 2^{32}-1$. For *bytevector-s32-set!* and *bytevector-s32-native-set!*, *n* must be an exact integer object in the interval $-2^{32,} \ldots, 2^{32}-1$.

These retrieve and set two-byte representations of numbers at indices *k, ..., k + 3*, according to the endianness specified by *endianness*. The procedures with *u32* in their names deal with the unsigned representation; those with *s32* in their names deal with the two's-complement representation.

The procedures with native in their names employ the native endianness, and work only at aligned indices: *k* must be a multiple of 4.

The ...*-set!* procedures return unspecified values.

3.4.6 Operation on 64-bit integers

Function: **bytevector-u64-ref** *bytevector k endianness*
Function: **bytevector-s64-ref** *bytevector k endianness*
Function: **bytevector-u64-native-ref** *bytevector k*
Function: **bytevector-s64-native-ref** *bytevector k*
Function: **bytevector-u64-set!** *bytevector k n endianness*
Function: **bytevector-s64-set!** *bytevector k n endianness*
Function: **bytevector-u64-native-set!** *bytevector k n*
Function: **bytevector-s64-native-set!** *bytevector k n*

[R6RS] *K, ... k + 7* must be a valid indices of *bytevector*. For *bytevector-u64-set!* and *bytevector-u64-native-set!*, *n* must be an exact integer object in the interval $0, \ldots, 2^{64}-1$. For *bytevector-s64-set!* and *bytevector-s64-native-set!*, *n* must be an exact integer object in the interval $-2^{64,} \ldots, 2^{64}-1$.

These retrieve and set two-byte representations of numbers at indices *k, ..., k + 7*, according to the endianness specified by *endianness*. The procedures with *u64* in their names deal with the unsigned representation; those with *s64* in their names deal with the two's-complement representation.

The procedures with native in their names employ the native endianness, and work only at aligned indices: *k* must be a multiple of 8.

The ...*-set!* procedures return unspecified values.

3.4.7 Operation on IEEE-754 representations

Function: **bytevector-ieee-single-native-ref** *bytevector k*
Function: **bytevector-ieee-single-ref** *bytevector k endianness*

[R6RS] *K, ..., k + 3* must be valid indices of *bytevector*. For *bytevector-ieee-single-native-ref*, *k*

must be a multiple of 4.

These procedures return the inexact real number object that best represents the IEEE-754 single-precision number represented by the four bytes beginning at index *k*.

Function: **bytevector-ieee-double-native-ref** *bytevector k*
Function: **bytevector-ieee-double-ref** *bytevector k endianness*

[R6RS] *K, ..., k + 7* must be valid indices of *bytevector*. For *bytevector-ieee-double-native-ref*, *k* must be a multiple of 8.

These procedures return the inexact real number object that best represents the IEEE-754 double-precision number represented by the four bytes beginning at index *k*.

Function: **bytevector-ieee-single-native-set!** *bytevector k x*
Function: **bytevector-ieee-single-set!** *bytevector k x endianness*

[R6RS] *K, ..., k + 3* must be valid indices of *bytevector*. For *bytevector-ieee-single-native-set!*, *k* must be a multiple of 4.

These procedures store an IEEE-754 single-precision representation of *x* into elements *k* through *k + 3* of *bytevector*, and return unspecified values.

Function: **bytevector-ieee-double-native-set!** *bytevector k x*
Function: **bytevector-ieee-double-set!** *bytevector k x endianness*

[R6RS] *K, ..., k + 7* must be valid indices of *bytevector*. For *bytevector-ieee-diuble-native-set!*, *k* must be a multiple of 8.

These procedures store an IEEE-754 double-precision representation of *x* into elements *k* through *k + 7* of *bytevector*, and return unspecified values.

3.4.8 Operation on strings

This section describes procedures that convert between strings and bytevectors containing Unicode encodings of those strings. When decoding bytevectors, encoding errors are handled as with the replace semantics of textual I/O: If an invalid or incomplete character encoding is encountered, then the replacement character U+FFFD is appended to the string being generated, an appropriate number of bytes are ignored, and decoding continues with the following bytes.

Function: **string->utf8** *string*

[R6RS] Returns a newly allocated (unless empty) bytevector that contains the UTF-8 encoding of the given string.

Function: **string->utf16** *string :optional endianness*

[R6RS] If *endianness* is specified, it must be the symbol big or the symbol little. The *string->utf16* procedure returns a newly allocated (unless empty) bytevector that contains the UTF-16BE or UTF-16LE encoding of the given string (with no byte-order mark). If endianness is not specified or is big, then UTF-16BE is used. If endianness is little, then UTF-16LE is used.

Function: **string->utf32** *string :optional endianness*

[R6RS] If *endianness* is specified, it must be the symbol big or the symbol little. The *string->utf32* procedure returns a newly allocated (unless empty) bytevector that contains the UTF-32BE or UTF-32LE encoding of the given string (with no byte mark). If endianness is not specified or is big, then UTF-32BE is used. If endianness is little, then UTF-32LE is used.

Function: **utf8->string** *bytevector*

[R6RS] Returns a newly allocated (unless empty) string whose character sequence is encoded by the given bytevector.

Function: **utf16->string** *bytevector endianness :optional endianness-mandatory?*

[R6RS] *Endianness* must be the symbol big or the symbol little. The *utf16->string* procedure returns a newly allocated (unless empty) string whose character sequence is encoded by the given bytevector. *Bytevector* is decoded according to UTF-16BE or UTF-16LE: If *endianness-mandatory?* is absent or #f, *utf16->string* determines the endianness according to a UTF-16 BOM at the beginning of *bytevector* if a BOM is present; in this case, the BOM is not decoded as a character. Also in this case, if no UTF-16 BOM is present, *endianness* specifies the endianness of the encoding. If *endianness-mandatory?* is a true value, *endianness* specifies the endianness of the encoding, and any UTF-16 BOM in the encoding is decoded as a regular character.

Function: **utf32->string** *bytevector endianness :optional endianness-mandatory?*

[R6RS] *Endianness* must be the symbol big or the symbol little. The *utf32->string* procedure returns a newly allocated (unless empty) string whose character sequence is encoded by the given bytevector. *Bytevector* is decoded according to UTF-32BE or UTF-32LE: If *endianness-mandatory?* is absent or #f, *utf32->string* determines the endianness according to a UTF-32 BOM at the beginning of *bytevector* if a BOM is present; in this case, the BOM is not decoded as a character. Also in this case, if no UTF-32 BOM is present, *endianness* specifies the endianness of the encoding. If *endianness-mandatory?* is a true value, *endianness* specifies the endianness of the encoding, and any UTF-32 BOM in the encoding is decoded as a regular character.

3.5 (rnrs lists (6))

Library: **(rnrs lists (6))**

[R6RS] The *(rnrs lists (6))*library, which contains various useful procedures that operate on lists.

Function: **find** *proc list*

[R6RS] *Proc* should accept one argument and return a single value. *Proc* should not mutate *list*. The *find* procedure applies *proc* to the elements of *list* in order. If *proc* returns a true value for an element, *find* immediately returns that element. If *proc* returns #f for all elements of the list, find returns #f.

Function: **for-all** *proc list1 list2 ...*

[R6RS+] Applies *proc* across each element of *lists*, and returns #f as soon as pred returns #f. If all application of *proc* return a non-false value, *for-all* returns the last result of the applications.

Function: **exists** *proc list1 list2 ...*

[R6RS+] Applies *proc* across each element of *lists*, and returns as soon as pred returns a non-false value. The return value of any is the non-false value pred returned. If *lists* are exhausted before *proc* returns a non-false value, #f is returned.

Note: R6RS requires the same length list for *for-all* and *exists.* On Sagittarius, however, these can accespt different length list and it will finish to process when the shortest list is finish to process.

Function: **filter** *proc list*

Function: **partition** *proc list*

[R6RS] *Proc* should accept one argument and return a single value.

The *filter* procedure applies *proc* to each element of *list* and returns a list of the elements of *list* for which *proc* returned a true value. The *partition* procedure also applies *proc* to each element of *list*, but returns two values, the first one a list of the elements of *list* for which *proc* returned a true value, and the second a list of the elements of *list* for which proc returned #f. In both cases, the elements of the result list(s) are in the same order as they appear in the input list. If multiple returns occur from *filter* or *partitions*, the return values returned by earlier returns are not mutated.

Function: **fold-left** *combine nil list1 list2 ...*

[R6RS+] *Combine* must be a procedure. It should accept one more argument than there are *lists*

and return a single value. It should not mutate the *list* arguments. The *fold-left* procedure iterates the *combine* procedure over an accumulator value and the elements of the *lists* from left to right, starting with an accumulator value of *nil*. More specifically, *fold-left* returns *nil* if the *lists* are empty. If they are not empty, *combine* is first applied to *nil* and the respective first elements of the *lists* in order. The result becomes the new accumulator value, and combine is applied to the new accumulator value and the respective next elements of the *list*. This step is repeated until the end of the list is reached; then the accumulator value is returned.

`Function:` **fold-right** *combine nil list1 list2 ...*

[R6RS+] *Combine* must be a procedure. It should accept one more argument than there are lists and return a single value. *Combine* should not mutate the list arguments. The *fold-right* procedure iterates the *combine* procedure over the elements of the *lists* from right to left and an accumulator value, starting with an accumulator value of *nil*. More specifically, *fold-right* returns *nil* if the *lists* are empty. If they are not empty, *combine* is first applied to the respective last elements of the *lists* in order and *nil*. The result becomes the new accumulator value, and combine is applied to the respective previous elements of the *lists* and the new accumulator value. This step is repeated until the beginning of the list is reached; then the accumulator value is returned.

Note: R6RS requires the same length list for *fold-left* and *fold-right.* On Sagittarius, however, these can accespt different length list and it will finish to process when the shortest list is finish to process.

`Function:` **remp** *proc list*
`Function:` **remove** *obj list*
`Function:` **remv** *obj list*
`Function:` **remq** *obj list*

[R6RS] *Proc* should accept one argument and return a single value. *Proc* should not mutate *list*.

Each of these procedures returns a list of the elements of *list* that do not satisfy a given condition. The *remp* procedure applies *proc* to each element of list and returns a list of the elements of *list* for which *proc* returned #f. The *remove*, *remv*, and *remq* procedures return a list of the elements that are not *obj*. The *remq* procedure uses *eq?* to compare *obj* with the elements of *list*, while *remv* uses *eqv?* and *remove* uses *equal?*. The elements of the result list are in the same order as they appear in the input list. If multiple returns occur from *remp*, the return values returned by earlier returns are not mutated.

`Function:` **memp** *proc list*
`Function:` **member** *obj list :optional =*
`Function:` **memv** *obj list*
`Function:` **memq** *obj list*

[R6RS+] *Proc* should accept one argument and return a single value. *Proc* should not mutate *list*.

These procedures return the first sublist of *list* whose car satisfies a given condition, where the sublists of *lists* are the lists returned by *(list-tail list k)* for *k* less than the length of *list*. The *memp* procedure applies *proc* to the cars of the sublists of *list* until it finds one for which *proc* returns a true value. The *member*, *memv*, and *memq* procedures look for the first occurrence of *obj*. If list does not contain an element satisfying the condition, then #f (not the empty list) is returned. The *member* procedure uses *equal?* or if = is given use it to compare *obj* with the elements of *list*, while *memv* uses *eqv?* and *memq* uses *eq?*.

`Function:` **assp** *proc alist*
`Function:` **assoc** *obj alist :optional =*
`Function:` **assv** *obj alist*
`Function:` **assq** *obj alist*

[R6RS+] *Alist* (for “association list”) should be a list of pairs. *Proc* should accept one argument

and return a single value. *Proc* should not mutate *alist*.

These procedures find the first pair in alist whose car field satisfies a given condition, and returns that pair without traversing *alist* further. If no pair in alist satisfies the condition, then #f is returned. The *assp* procedure successively applies *proc* to the car fields of alist and looks for a pair for which it returns a true value. The *assoc*, *assv*, and *assq* procedures look for a pair that has *obj* as its car. The *assoc* procedure uses *equal?* or if = is given use it to compare *obj* with the car fields of the pairs in *alist*, while *assv* uses *eqv?* and *assq* uses *eq?*.

Note: *member* and *assoc* procedures are the same behaviour as SRFI-1.

`Function:` **cons*** *obj1 obj2 ...*

[R6RS] Like *list*, but the last argument provides the tail of the constructed list.

3.6 (rnrs sorting (6))

The *(rnrs sorting (6))* library for sorting lists and vectors.

`Function:` **list-sort** *proc list*
`Function:` **vector-sort** *proc vector*

[R6RS] *Proc* should accept any two elements of *list* or *vector*, and should not have any side effects. *Proc* should return a true value when its first argument is strictly less than its second, and #f otherwise.

The *list-sort* and *vector-sort* procedures perform a stable sort of *list* or *vector* in ascending order according to *proc*, without changing *list* or *vector* in any way. The *list-sort* procedure returns a list, and *vector-sort* returns a vector. The results may be *eq?* to the argument when the argument is already sorted, and the result of *list-sort* may share structure with a tail of the original list. The sorting algorithm performs O(n lg n) calls to *proc* where n is the length of *list* or *vector*, and all arguments passed to *proc* are elements of the list or vector being sorted, but the pairing of arguments and the sequencing of calls to *proc* are not specified. If multiple returns occur from *list-sort* or *vector-sort*, the return values returned by earlier returns are not mutated.

`Function:` **vector-sort!** *proc vector*

[R6RS] *Proc* should accept any two elements of the *vector*, and should not have any side effects. *Proc* should return a true value when its first argument is strictly less than its second, and #f otherwise. The *vector-sort!* procedure destructively sorts *vector* in ascending order according to *proc*. The sorting algorithm performs O(n2) calls to *proc* where n is the length of *vector*, and all arguments passed to *proc* are elements of the vector being sorted, but the pairing of arguments and the sequencing of calls to proc are not specified. The sorting algorithm may be unstable. The procedure returns unspecified values.

3.7 (rnrs control (6))

The *(rnrs control (6))* library, which provides useful control structures.

`Syntax:` **when** *test expression ...*
`Syntax:` **unless** *test expression ...*

[R6RS] *Test* must be an expression.

A *when* expression is evaluated by evaluating the *test* expression. If *test* evaluates to a true value, the remaining *expression*s are evaluated in order, and the results of the last *expression* are returned as the results of the entire *when* expression. Otherwise, the *when* expression returns unspecified values. An *unless* expression is evaluated by evaluating the *test* expression. If *test* evaluates to #f, the remaining *expression*s are evaluated in order, and the results of the last *expression* are returned as the results of the entire *unless* expression. Otherwise, the *unless* expression returns unspecified values.

Syntax:**do** *((variable init step) ...) (test expression ...) command*

[R6RS] The *init*s, *step*s, *test*s, and *command*s must be expressions. The *variable*s must be pairwise distinct variables.

The *do* expression is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration.

A *do* expression is evaluated as follows: The *init* expressions are evaluated (in some unspecified order), the *variable*s are bound to fresh locations, the results of the *init* expressions are stored in the bindings of the *variable*s, and then the iteration phase begins.

Each iteration begins by evaluating *test*> if the result is #f, then the *command*s are evaluated in order for effect, the *step* expressions are evaluated in some unspecified order, the *variable*s are bound to fresh locations holding the results, and the next iteration begins.

If *test* evaluates to a true value, the *expression*s are evaluated from left to right and the values of the last *expression* are returned. If no *expression*s are present, then the do expression returns unspecified values.

The regionof the binding of a *variable* consists of the entire do expression except for the *init*s.

A *step* may be omitted, in which case the effect is the same as if (*variable init variable*) had been written instead of (*variable init*).

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))          ⇒ #(0 1 2 3 4)


(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))           ⇒ 25
```

Macro:**case-lambda** *calse-lambda-clause ...*

[R6RS] Each *case-lambda-clause* must be of the form

(*formals body*)

Formals must be as in a *lambda* form.

A *case-lambda* expression evaluates to a procedure. This procedure, when applied, tries to match its arguments to the *case-lambda-clause*s in order. The arguments match a clause if one of the following conditions is fulfilled:

Formals has the form (*variable ...*) and the number of arguments is the same as the number of formal parameters in *formals*.

Formals has the form

(*variable1 ...variablen . variablen+1>*

and the number of arguments is at least *n*.

Formals has the form *variable*.

For the first clause matched by the arguments, the variables of the *formals* are bound to fresh locations containing the argument values in the same arrangement as with lambda.

The last expression of a *body* in a case-lambda expression is in tail context.

If the arguments match none of the clauses, an exception with condition type *&assertion* is raised.

3.8 (rnrs records syntactic (6))

The *(rnrs records syntactic (6))* library. Some details of the specification are explained in terms of the specification of the procedural layer below.

Macro:**define-record-type** *name-spec record-clause…*
Auxiliary syntax:**fields** *field-spec …*
Auxiliary syntax:**parent** *parent-name*
Auxiliary syntax:**protocol** *expression*
Auxiliary syntax:**sealed** *boolean*
Auxiliary syntax:**opaque** *boolean*
Auxiliary syntax:**nongenerative** *:optional uid*
Auxiliary syntax:**parent-rtd** *parent-rtd parent-cd*

[R6RS] A *define-record-type* form defines a record type along with associated constructor descriptor and constructor, predicate, field accessors, and field mutators. The *define-record-type* form expands into a set of definitions in the environment where define-record-type appears; hence, it is possible to refer to the bindings (except for that of the record type itself) recursively.

The *name-spec* specifies the names of the record type, constructor, and predicate. It must take one of the following forms:

(record-name constructor-name predicate-name)

record-name

Record-name, *constructor-name*, and *predicate-name* must all be identifiers.

Record-name>, taken as a symbol, becomes the name of the record type. (See the description of *make-record-type-descriptor*.) Additionally, it is bound by this definition to an expand-time or run-time representation of the record type and can be used as parent name in syntactic record-type definitions that extend this definition. It can also be used as a handle to gain access to the underlying record-type descriptor and constructor descriptor (see r*ecord-type-descriptor* and *record-constructor-descriptor*).

Constructor-name> is defined by this definition to be a constructor for the defined record type, with a protocol specified by the protocol clause, or, in its absence, using a default protocol. For details, see the description of the *protocol* clause below.

Predicate-name is defined by this definition to a predicate for the defined record type.

The second form of *name-spec* is an abbreviation for the first form, where the name of the constructor is generated by prefixing the record name with *make-*, and the predicate name is generated by adding a question mark (*?*) to the end of the record name. For example, if the record name is *frob*, the name of the constructor is *make-frob*, and the predicate name is *frob?*.

Each *record-clause* must take one of the auxiliary syntax forms; it is a syntax violation if multiple *record-clause*s of the same kind appear in a *define-record-type* form.

(fields field-spec…)

Each *field spec* has one of the following forms

(immutable field-name accessor-name)

(mutable field-name accessor-name mutator-name)

(immutable field-name)

(mutable field-name)

field-name

Field-name>, *accessor-name*, and *mutator-name*> must all be identifiers. The first form declares an immutable field called *field-name*>, with the corresponding accessor named *accessor-name*. The second form declares a mutable field called *field-name*, with the corresponding accessor named *accessor-name*, and with the corresponding mutator named *mutator-name*.

If *field-spec* takes the third or fourth form, the accessor name is generated by appending the record name and field name with a hyphen separator, and the mutator name (for a mutable field) is generated by adding a *-set!* suffix to the accessor name. For example, if the record name is *frob* and the field name is *widget*, the accessor name is *frob-widget* and the mutator name is *frob-widget-set!*.

If *field-spec* is just a *field-name* form, it is an abbreviation for *(immutable field-name)*.

The *field-name*s become, as symbols, the names of the fields in the record-type descriptor being created, in the same order.

The *fields* clause may be absent; this is equivalent to an empty *fields* clause.

(parent parent-name)

Specifies that the record type is to have parent type *parent-name*, where *parent-name* is the *record-name* of a record type previously defined using *define-record-type*. The record-type definition associated with *parent-name* must not be sealed. If no *parent* clause and no *parent-rtd* (see below) clause is present, the record type is a base type.

(protocol expression)

Expression is evaluated in the same environment as the *define-record-type* form, and must evaluate to a protocol appropriate for the record type being defined.

The protocol is used to create a record-constructor descriptor as described below. If no *protocol* clause is specified, a constructor descriptor is still created using a default protocol. The clause can be absent only if the record type being defined has no parent type, or if the parent definition does not specify a protocol.

(sealed boolean)

If this option is specified with operand #t, the defined record type is sealed, i.e., no extensions of the record type can be created. If this option is specified with operand #f, or is absent, the defined record type is not sealed.

(opaque boolean)

If this option is specified with operand #t, or if an opaque parent record type is specified, the defined record type is opaque. Otherwise, the defined record type is not opaque. See the specification of record-rtd below for details.

(nongenerative uid)

(nongenerative)

This specifies that the record type is nongenerative with uid *uid*, which must be an *identifier*. If *uid* is absent, a unique uid is generated at macro-expansion time. If two record-type definitions specify the same uid, then the record-type definitions should be equivalent, i.e., the implied arguments to *make-record-type-descriptor* must be equivalent as described under *make-record-type-descriptor*. If this condition is not met, it is either considered a syntax violation or an exception with condition type *&assertion* is raised. If the condition is met, a single record type is generated for both definitions.

In the absence of a *nongenerative* clause, a new record type is generated every time a define-record-type form is evaluated:

```
(let ((f (lambda (x)
           (define-record-type r ...)
           (if x r? (make-r ...)))))
  ((f #t) (f #f)))        ⇒ #f
```

(parent-rtd parent-rtd parent-cd)

Specifies that the record type is to have its parent type specified by *parent-rtd*, which should be an expression evaluating to a record-type descriptor, and *parent-cd*, which should be an expression evaluating to a constructor descriptor. The record-type definition associated with the value of *parent-rtd* must not be sealed. Moreover, a record-type definition must not have both a *parent* and a *parent-rtd* clause.

All bindings created by define-record-type (for the record type, the constructor, the predicate, the accessors, and the mutators) must have names that are pairwise distinct.

The constructor created by a define-record-type form is a procedure as follows:

- If there is no parent clause and no protocol clause, the constructor accepts as many arguments as there are fields, in the same order as they appear in the fields clause, and returns a record object with the fields initialized to the corresponding arguments.
- If there is no parent or parent-rtd clause and a protocol clause, the protocol expression must evaluate to a procedure that accepts a single argument. The protocol procedure is called once during the evaluation of the define-record-type form with a procedure p as its argument. It should return a procedure, which will become the constructor bound to <constructor name>. The procedure p accepts as many arguments as there are fields, in the same order as they appear in the fields clause, and returns a record object with the fields initialized to the corresponding arguments.

  The constructor returned by the protocol procedure can accept an arbitrary number of arguments, and should call p once to construct a record object, and return that record object.

  For example, the following protocol expression for a record-type definition with three fields creates a constructor that accepts values for all fields, and initialized them in the reverse order of the arguments:

  ```
  (lambda (p)
    (lambda (v1 v2 v3)
      (p v3 v2 v1)))
  ```

- If there is both a parent clause and a protocol clause, then the protocol procedure is called once with a procedure n as its argument. As in the previous case, the protocol procedure should return a procedure, which will become the constructor bound to <constructor name>. However, n is different from p in the previous case: It accepts arguments corresponding to the arguments of the constructor of the parent type. It then returns a procedure p that accepts as many arguments as there are (additional) fields in this type, in the same order as in the fields clause, and returns a record object with the fields of the parent record types initialized according to their constructors and the arguments to n, and the fields of this record type initialized to its arguments of p.

  The constructor returned by the protocol procedure can accept an arbitrary number of arguments, and should call n once to construct the procedure p, and call p once to create the record object, and finally return that record object.

For example, the following protocol expression assumes that the constructor of the parent type takes three arguments:

```
(lambda (n)
  (lambda (v1 v2 v3 x1 x2 x3 x4)
    (let ((p (n v1 v2 v3)))
      (p x1 x2 x3 x4))))
```

The resulting constructor accepts seven arguments, and initializes the fields of the parent types according to the constructor of the parent type, with v1, v2, and v3 as arguments. It also initializes the fields of this record type to the values of x1, ..., x4.

- If there is a parent clause, but no protocol clause, then the parent type must not have a protocol clause itself. The constructor bound to *constructor-name* is a procedure that accepts arguments corresponding to the parent types' constructor first, and then one argument for each field in the same order as in the fields clause. The constructor returns a record object with the fields initialized to the corresponding arguments.
- If there is a parent-rtd clause, then the constructor is as with a parent clause, except that the constructor of the parent type is determined by the constructor descriptor of the parent-rtd clause.

Macro: **record-type-descriptor** *record-name*

[R6RS] Evaluates to the record-type descriptor (see below) associated with the type specified by *record-name*.

Macro: **record-constructor-descriptor** *record-name*

[R6RS] Evaluates to the record-type constructor (see below) associated with the type specified by *record-name*.

The following example uses the record? procedure from the *(rnrs records inspection (6))* library:

```
(define-record-type (point make-point point?)
  (fields (immutable x point-x)
          (mutable y point-y set-point-y!))
  (nongenerative point-4893d957-e00b-11d9-817f-00111175eb9e))
(define-record-type (cpoint make-cpoint cpoint?)
  (parent point)
  (protocol (lambda (n)
              (lambda (x y c)
                ((n x y) (color->rgb c)))))
  (fields (mutable rgb cpoint-rgb cpoint-rgb-set!)))
(define (color->rgb c) (cons 'rgb c))
(define p1 (make-point 1 2))
(define p2 (make-cpoint 3 4 'red))
(point? p1)        ⇒#t
(point? p2)        ⇒#t
(point? (vector))        ⇒#f
```

```
(point? (cons 'a 'b))   ⇒#f
(cpoint? p1)        ⇒#f
(cpoint? p2)        ⇒#t
(point-x p1)        ⇒1
(point-y p1)        ⇒2
(point-x p2)        ⇒3
(point-y p2)        ⇒4
(cpoint-rgb p2)    ⇒(rgb . red)
(set-point-y! p1 17)        ⇒unspecified
(point-y p1)        ⇒17)
(record-rtd p1)    ⇒(record-type-descriptor point)
(define-record-type (ex1 make-ex1 ex1?)
  (protocol (lambda (p) (lambda a (p a))))
  (fields (immutable f ex1-f)))
(define ex1-i1 (make-ex1 1 2 3))
(ex1-f ex1-i1)        ⇒(1 2 3)
(define-record-type (ex2 make-ex2 ex2?)
  (protocol
    (lambda (p) (lambda (a . b) (p a b))))
  (fields (immutable a ex2-a)
          (immutable b ex2-b)))
(define ex2-i1 (make-ex2 1 2 3))
(ex2-a ex2-i1)        ⇒1
(ex2-b ex2-i1)        ⇒(2 3)
(define-record-type (unit-vector make-unit-vector unit-vector?)
  (protocol (lambda (p)
              (lambda (x y z)
                (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
                     (p (/ x length) (/ y length) (/ z length))))))
  (fields (immutable x unit-vector-x)
          (immutable y unit-vector-y)
          (immutable z unit-vector-z)))
(define *ex3-instance* #f)
(define-record-type ex3
  (parent cpoint)
  (protocol (lambda (n)
              (lambda (x y t)
```

```
                (let ((r ((n x y 'red) t)))
                  (set! *ex3-instance* r)
                  r))))
    (fields  (mutable thickness))
    (sealed #t) (opaque #t))
  (define ex3-i1 (make-ex3 1 2 17))
  (ex3? ex3-i1)        ⇒#t
  (cpoint-rgb ex3-i1)        ⇒(rgb . red)
  (ex3-thickness ex3-i1)        ⇒17
  (ex3-thickness-set! ex3-i1 18)  ⇒unspecified
  (ex3-thickness ex3-i1)        ⇒18
  *ex3-instance*        ⇒ex3-i1
  (record? ex3-i1)        ⇒#f
```

3.9 (rnrs records procedural (6))

The procedural layer is provided by the *(rnrs records procedural (6))* library.

`Function:` **make-record-type-descriptor** *name parent uid sealed? opaque? fields*

[R6RS] Returns a record-type descriptor (rtd).

The *name* argument must be a symbol. It names the record type, and is intended purely for informational purposes and may be used for printing by the underlying Scheme system.

The *parent* argument must be either #f or an rtd. If it is an rtd, the returned record type, *t*, extends the record type *p* represented by parent. An exception with condition type *&assertion* is raised if parent is sealed (see below).

The *uid* argument must be either #f or a symbol. If *uid* is a symbol, the record-creation operation is *nongenerative* i.e., a new record type is created only if no previous call to *make-record-type-descriptor* was made with the *uid*. If *uid* is #f, the record-creation operation is generative, i.e., a new record type is created even if a previous call to *make-record-type-descriptor* was made with the same arguments.

If *make-record-type-descriptor* is called twice with the same *uid* symbol, the parent arguments in the two calls must be *eqv?*, the *fields* arguments *equal?*, the *sealed?* arguments boolean-equivalent (both #f or both true), and the *opaque?* arguments boolean-equivalent. If these conditions are not met, an exception with condition type *&assertion* is raised when the second call occurs. If they are met, the second call returns, without creating a new record type, the same record-type descriptor (in the sense of *eqv?*) as the first call.

The *sealed?* flag must be a boolean. If true, the returned record type is sealed, i.e., it cannot be extended.

The *opaque?* flag must be a boolean. If true, the record type is opaque. If passed an instance of the record type, *record?* returns #f. Moreover, if *record-rtd* (see *(rnrs records inspection (6))*) is called with an instance of the record type, an exception with condition type *&assertion* is raised. The record type is also opaque if an opaque parent is supplied. If *opaque?* is #f and an opaque parent is not supplied, the record is not opaque.

The *fields* argument must be a vector of field specifiers. Each field specifier must be a list of the form (mutable *name*) or a list of the form (immutable *name*). Each name must be a symbol and

names the corresponding field of the record type; the names need not be distinct. A field identified as mutable may be modified, whereas, when a program attempts to obtain a mutator for a field identified as immutable, an exception with condition type *&assertion* is raised. Where field order is relevant, e.g., for record construction and field access, the fields are considered to be ordered as specified, although no particular order is required for the actual representation of a record instance.

The specified fields are added to the parent fields, if any, to determine the complete set of fields of the returned record type. If *fields* is modified after *make-record-type-descriptor* has been called, the effect on the returned rtd is unspecified.

A *generative* record-type descriptor created by a call to *make-record-type-descriptor* is not *eqv?* to any record-type descriptor (generative or nongenerative) created by another call to *make-record-type-descriptor*. A generative record-type descriptor is *eqv?* only to itself, i.e., (*eqv? rtd1 rtd2*) iff (*eq? rtd1 rtd2*). Also, two nongenerative record-type descriptors are *eqv?* if they were created by calls to *make-record-type-descriptor* with the same uid arguments.

`Function:` **record-type-descriptor?** *obj*

[R6RS] Returns #t if the argument is a record-type descriptor, #f otherwise.

`Function:` **make-record-constructor-descriptor** *rtd parent-constructor-descriptor protocol*

[R6RS] Returns a *record-constructor descriptor* (or *constructor descriptor* for short) that specifies a *record constructor* (or *constructor* for short), that can be used to construct record values of the type specified by *rtd*, and which can be obtained via *record-constructor*. A constructor descriptor can also be used to create other constructor descriptors for subtypes of its own record type. *Rtd* must be a record-type descriptor. *Protocol* must be a procedure or #f. If it is #f, a default protocol procedure is supplied.

If *protocol* is a procedure, it is handled analogously to the protocol expression in a *define-record-type* form.

If *rtd* is a base record type and protocol is a procedure, *parent-constructor-descriptor* must be #f. In this case, *protocol* is called by *record-constructor* with a single argument *p*. *P* is a procedure that expects one argument for every field of *rtd* and returns a record with the fields of *rtd* initialized to these arguments. The procedure returned by *protocol* should call *p* once with the number of arguments p expects and return the resulting record as shown in the simple example below:

```
(lambda (p)
  (lambda (v1 v2 v3)
    (p v1 v2 v3)))
```

Here, the call to *p* returns a record whose fields are initialized with the values of *v1*, *v2*, and *v3*. The expression above is equivalent to (lambda (p) p). Note that the procedure returned by *protocol* is otherwise unconstrained; specifically, it can take any number of arguments.

If *rtd* is an extension of another record type *parent-rtd* and *protocol* is a procedure, *parent-constructor-descriptor* must be a constructor descriptor of *parent-rtd* or #f. If *parent-constructor-descriptor* is a constructor descriptor, *protocol* it is called by *record-constructor* with a single argument n, which is a procedure that accepts the same number of arguments as the constructor of *parent-constructor-descriptor* and returns a procedure p that, when called, constructs the record itself. The *p* procedure expects one argument for every field of *rtd* (not including parent fields) and returns a record with the fields of *rtd* initialized to these arguments, and the fields of *parent-rtd* and its parents initialized as specified by *parent-constructor-descriptor*.

The procedure returned by *protocol* should call *n* once with the number of arguments *n* expects, call the procedure *p* it returns once with the number of arguments p expects and return the resulting record. A simple *protocol* in this case might be written as follows:

```
(lambda (n)
  (lambda (v1 v2 v3 x1 x2 x3 x4)
    (let ((p (n v1 v2 v3)))
      (p x1 x2 x3 x4))))
```

This passes arguments *v1, v2, v3* to n for *parent-constructor-descriptor* and calls *p* with *x1, ..., x4* to initialize the fields of *rtd* itself.

Thus, the constructor descriptors for a record type form a sequence of protocols parallel to the sequence of record-type parents. Each constructor descriptor in the chain determines the field values for the associated record type. Child record constructors need not know the number or contents of parent fields, only the number of arguments accepted by the parent constructor.

Protocol may be #f, specifying a default constructor that accepts one argument for each field of *rtd* (including the fields of its parent type, if any). Specifically, if *rtd* is a base type, the default protocol procedure behaves as if it were (lambda (p) p). If *rtd* is an extension of another type, then *parent-constructor-descriptor* must be either #f or itself specify a default constructor, and the default *protocol* procedure behaves as if it were:

```
(lambda (n)
  (lambda (v1 ... vj x1 ... xk)
    (let ((p (n v1 ... vj)))
      (p x1 ... xk))))
```

The resulting constructor accepts one argument for each of the record type's complete set of fields (including those of the parent record type, the parent's parent record type, etc.) and returns a record with the fields initialized to those arguments, with the field values for the parent coming before those of the extension in the argument list. (In the example, *j* is the complete number of fields of the parent type, and k is the number of fields of *rtd* itself.)

If *rtd* is an extension of another record type, and *parent-constructor-descriptor* or the *protocol* of *parent-constructor-descriptor* is #f, *protocol* must also be #f, and a default constructor descriptor as described above is also assumed.

`Function:` **record-constructor** *constructor-descriptor*

[R6RS] Calls the *protocol* of *constructor-descriptor* (as described for *make-record-constructor-descriptor*) and returns the resulting constructor *constructor* for records of the record type associated with *constructor-descriptor.*

`Function:` **record-predicate** *rtd*

[R6RS] Returns a procedure that, given an object *obj*, returns #t if *obj* is a record of the type represented by *rtd*, and #f otherwise.

`Function:` **record-accessor** *rtd k*

[R6RS] *K* must be a valid field index of *rtd*. The *record-accessor* procedure returns a one-argument procedure whose argument must be a record of the type represented by *rtd*. This procedure returns the value of the selected field of that record.

The field selected corresponds to the *k*th element (0-based) of the *fields* argument to the invocation of *make-record-type-descriptor* that created rtd. Note that *k* cannot be used to specify a field of any type *rtd* extends.

`Function:` **record-mutator** *rtd k*

[R6RS] *K* must be a valid field index of *rtd*. The r*ecord-mutator* procedure returns a two-argument procedure whose arguments must be a record record *r* of the type represented by *rtd* and an object *obj*. This procedure stores *obj* within the field of r specified by *k*. The *k* argument is as in

record-accessor. If *k* specifies an immutable field, an exception with condition type *&assertion* is raised. The mutator returns unspecified values.

3.10 (rnrs records inspection (6))

The *(rnrs records inspection (6))* library provides procedures for inspecting records and their record-type descriptors. These procedures are designed to allow the writing of portable printers and inspectors.

`Function:` **record?** *obj*
[R6RS] Returns #t if obj is a record, and its record type is not opaque, and returns #f otherwise.

`Function:` **record-rtd** *record*
[R6RS] Returns the rtd representing the type of *record* if the type is not opaque. The rtd of the most precise type is returned; that is, the type *t* such that *record* is of type t but not of any type that extends *t*. If the type is opaque, an exception is raised with condition type *&assertion.*

`Function:` **record-type-name** *rtd*
[R6RS] Returns the name of the record-type descriptor *rtd*.

`Function:` **record-type-parent** *rtd*
[R6RS] Returns the parent of the record-type descriptor *rtd*, or #f if it has none.

`Function:` **record-type-uid** *rtd*
[R6RS] Returns the uid of the record-type descriptor *rtd*, or #f if it has none.

`Function:` **record-type-generative?** *rtd*
[R6RS] Returns #t if *rtd* is generative, and #f if not.

`Function:` **record-type-sealed?** *rtd*
[R6RS] Returns #t if the record-type descriptor is sealed, and #f if not.

`Function:` **record-type-opaque?** *rtd*
[R6RS] Returns #t if the the record-type descriptor is opaque, and #f if not.

`Function:` **record-type-field-names** *rtd*
[R6RS] Returns a vector of symbols naming the fields of the type represented by *rtd* (not including the fields of parent types) where the fields are ordered as described under *make-record-type-descriptor.*

`Function:` **record-type-mutable?** *rtd k*
[R6RS] Returns #t if the field specified by *k* of the type represented by *rtd* is mutable, and #f if not. *K* is as in *record-accessor.*

3.11 (rnrs exceptions (6))

This section describes exception-handling and exception-raising constructs provided by the *(rnrs exceptions (6))* library.

`Function:` **with-exception-handler** *handler thunk*
[R6RS] *Handler* must be a procedure and should accept one argument. *Thunk* must be a procedure that accepts zero arguments. The *with-exception-handler* procedure returns the results of invoking *thunk*. When an exception is raised, *handler* will be invoked with the exception.

`Macro:` **guard** *(veriable cond-clause ...) body*
[R6RS] Each *cond-clause* is as in the specification of *cond*. and *else* are the same as in the *(rnrs base (6))* library.

Evaluating a guard form evaluates *body* with an exception handler that binds the raised object to

variable and within the scope of that binding evaluates the clauses as if they were the clauses of a cond expression. If every *cond-clause*'s *test* evaluates to #f and there is no *else* clause, then raise is re-invoked on the raised object.

`Function:` **raise** *obj*
[R6RS] Raises a non-continuable exception by invoking the current exception handler on *obj*.

`Function:` **raise-continuable** *obj*
[R6RS] Raises a *continuable exception* by invoking the current exception handler on *obj*.

3.12 (rnrs conditions (6))

The section describes *(rnrs conditions (6))* library for creating and inspecting condition types and values. A condition value encapsulates information about an exceptional situation.

`Function:` **condition** *condition ...*
[R6RS] The *condition* procedure returns a condition object with the components of the *conditions* as its components, in the same order. The returned condition is compound if the total number of components is zero or greater than one. Otherwise, it may be compound or simple.

`Function:` **simple-conditions** *condition*
[R6RS] The *simple-conditions* procedure returns a list of the components of condition, in the same order as they appeared in the construction of *condition.*

`Function:` **condition?** *obj*
[R6RS] Returns #t if obj is a (simple or compound) condition, otherwise returns #f.

`Function:` **condition-predicate** *rtd*
[R6RS] *Rtd* must be a record-type descriptor of a subtype of *&condition*. The *condition-predicate* procedure returns a procedure that takes one argument. This procedure returns #t if its argument is a condition of the condition type represented by *rtd*, i.e., if it is either a simple condition of that record type (or one of its subtypes) or a compound condititon with such a simple condition as one of its components, and #f otherwise.

`Function:` **condition-accessor** *rtd proc*
[R6RS] *Rtd* must be a record-type descriptor of a subtype of *&condition*. *Proc* should accept one argument, a record of the record type of *rtd*. The *condition-accessor* procedure returns a procedure that accepts a single argument, which must be a condition of the type represented by *rtd*. This procedure extracts the first component of the condition of the type represented by *rtd*, and returns the result of applying *proc* to that component.

`Macro:` **define-condition-type** *condition-type supertypes constructor predicate field-spec ...*
[R6RS] *Condition-type*, *supertypes*, *constructor*, and *predicate* must all be identifiers. Each *field-spec* must be of the form

(*field accessor*)

where both *field* and *accessor* must be identifiers.

The *define-condition-type* form expands into a record-type definition for a record type *condition-type*. The record type will be non-opaque, non-sealed, and its fields will be immutable. It will have *supertype* has its parent type. The remaining identifiers will be bound as follows:

- *Constructor* is bound to a default constructor for the type : It accepts one argument for each of the record type's complete set of fields (including parent types, with the fields of the parent coming before those of the extension in the arguments) and returns a condition object initialized to those arguments.
- *Predicate* is bound to a predicate that identifies conditions of type *condition-type* or any of

its subtypes.

- Each *accessor* is bound to a procedure that extracts the corresponding field from a condition of type *condition-type*.

3.12.1 Standard condition types

Hierarchy of standard condition types:

- &condition
  - &warning
  - &serious
    - &error
    - &violation
      - &assertion
      - &non-continuable
      - &implementation-restriction
      - &lexical
      - &syntax
      - &undefined
  - &message
  - &irritants

Condition Type:**&message**
Function:**make-message-condition** *message*
Function:**message-condition?** *obj*
Function:**condition-message** *condition*

[R6RS] It carries a message further describing the nature of the condition to humans.

Condition Type:**&warning**
Function:**make-warning**
Function:**warning?** *obj*

[R6RS] This type describes conditions that do not, in principle, prohibit immediate continued execution of the program, but may interfere with the program's execution later.

Condition Type:**&serious**
Function:**make-serious-condition**
Function:**serious-condition?** *obj*

[R6RS] This type describes conditions serious enough that they cannot safely be ignored. This condition type is primarily intended as a supertype of other condition types.

Condition Type:**&error**
Function:**make-error**
Function:**error?** *obj*

[R6RS] This type describes errors, typically caused by something that has gone wrong in the interaction of the program with the external world or the user.

Condition Type:**&violation**
Function:**make-violation**

Function: **violation?** *obj*

[R6RS] This type describes violations of the language standard or a library standard, typically caused by a programming error.

Condition Type: **&assertion**
Function: **make-assertion-violation**
Function: **assertion-violation?** *obj*

[R6RS] This type describes an invalid call to a procedure, either passing an invalid number of arguments, or passing an argument of the wrong type.

Condition Type: **&irritants**
Function: **make-irritants-condition** *irritants*
Function: **irritants-condition?** *obj*
Function: **condition-irritants** *condition*

[R6RS] *Irritants* should be a list of objects. This condition provides additional information about a condition, typically the argument list of a procedure that detected an exception. Conditions of this type are created by the *error* and *assertion-violation* procedures.

Condition Type: **&who**
Function: **make-who-condition** *who*
Function: **who-condition?** *obj*
Function: **condition-who** *condition*

[R6RS] Who should be a symbol or string identifying the entity reporting the exception. Conditions of this type are created by the *error* and *assertion-violation* procedures, and the *syntax-violation* procedure.

Condition Type: **&non-continuable**
Function: **make-non-continuable-violation**
Function: **non-continuable-violation?** *obj*

[R6RS] This type indicates that an exception handler invoked via raise has returned.

Condition Type: **&implementation-restriction**
Function: **make-implementation-restriction-violation**
Function: **implementation-restriction-violation?** *obj*

[R6RS] This type describes a violation of an implementation restriction allowed by the specification, such as the absence of representations for NaNs and infinities.

Condition Type: **&lexical**
Function: **make-lexical-violation**
Function: **lexical-violation?** *obj*

[R6RS] This type describes syntax violations at the level of the datum syntax.

Condition Type: **&syntax**
Function: **make-syntax-violation** *irritants*
Function: **syntax-violation?** *obj*
Function: **syntax-violation-form** *condition*
Function: **syntax-violation-subform** *condition*

[R6RS] This type describes syntax violations. *Form* should be the erroneous syntax object or a datum representing the code of the erroneous form. *Subform* should be an optional syntax object or datum within the erroneous form that more precisely locates the violation. It can be #f to indicate the absence of more precise information.

Condition Type: **&undefined**
Function: **make-undefined-violation**
Function: **undefined-violation?** *obj*

[R6RS] This type describes unbound identifiers in the program.

3.13 (rnrs io ports) and (rnrs io simple)

3.13.1 Condition types

The condition types and corresponding predicates and accessors are exported by both the *(rnrs io ports (6))* and *(rnrs io simple (6))* libraries. They are also exported by the *(rnrs files (6))* library.

Condition Type: **&i/o**
Function: **make-i/o-error**
Function: **i/o-error?** *obj*
[R6RS] This is a supertype for a set of more specific I/O errors.

Condition Type: **&i/o-read**
Function: **make-i/o-read-error**
Function: **i/o-read-error?** *obj*
[R6RS] This condition type describes read errors that occurred during an I/O operation.

Condition Type: **&i/o-write**
Function: **make-i/o-write-error**
Function: **i/o-write-error?** *obj*
[R6RS] This condition type describes write errors that occurred during an I/O operation.

Condition Type: **&i/o-invalid-position**
Function: **make-i/o-invalid-position-error** *position*
Function: **i/o-invalid-position-error?** *obj*
Function: **i/o-error-position** *condition*
[R6RS] This condition type describes attempts to set the file position to an invalid position. *Position* should be the file position that the program intended to set. This condition describes a range error, but not an assertion violation.

Condition Type: **&i/o-filename**
Function: **make-i/o-filename-error** *filename*
Function: **i/o-filename-error?** *obj*
Function: **i/o-error-filename** *condition*
[R6RS] This condition type describes an I/O error that occurred during an operation on a named file. *Filename* should be the name of the file.

Condition Type: **&i/o-file-protection**
Function: **make-i/o-file-protection-error** *filename*
Function: **i/o-file-protection-error?** *obj*
[R6RS] A condition of this type specifies that an operation tried to operate on a named file with insufficient access rights.

Condition Type: **&i/o-file-is-read-only**
Function: **make-i/o-file-is-read-only-error** *filename*
Function: **i/o-file-is-read-only-error?** *obj*
[R6RS] A condition of this type specifies that an operation tried to operate on a named read-only file under the assumption that it is writeable.

Condition Type: **&i/o-file-already-exists**
Function: **make-i/o-file-already-exists-error** *filename*
Function: **i/o-file-already-exists-error?** *obj*
[R6RS] A condition of this type specifies that an operation tried to operate on an existing named

file under the assumption that it did not exist.

Condition Type: **&i/o-file-does-not-exist**
Function: **make-i/o-file-does-not-exist-error** *filename*
Function: **i/o-file-does-not-exist-error?** *obj*

[R6RS] A condition of this type specifies that an operation tried to operate on an non-existent named file under the assumption that it existed.

Condition Type: **&i/o-port**
Function: **make-i/o-port-error** *port*
Function: **i/o-port-error?** *obj*
Function: **i/o-error-port** *condition*

[R6RS] This condition type specifies the port with which an I/O error is associated. *Port* should be the port. Conditions raised by procedures accepting a port as an argument should include an &i/o-port-error condition.

3.13.2 Filename

In this world, unfortunately there are a lot of operating systems. However, as far as I know there are only two file separators, one is Windows style back slash, the other is UNIX style shash. On Sagittarius both of it can be used as file path.Inside of the resolution of file path, Sagittarius replaces those file separators to OS specific one. Even though programmer does not have to care about it, I think it's better to use slash as file seperator on script files.

Macro: **file-options** *file-option-symbol ...*

[R6RS] Each *file-option-symbol* must be a symbol. The *file-options* syntax returns a file-options object that encapsulates the specified options.

When supplied to an operation that opens a file for output, the file-options object returned by (*file-options*) specifies that the file is created if it does not exist and an exception with condition type *&i/o-file-already-exists* is raised if it does exist. The following standard options can be included to modify the default behavior.

- *no-create* If the file does not already exist, it is not created; instead, an exception with condition type *&i/o-file-does-not-exist* is raised. If the file already exists, the exception with condition type *&i/o-file-already-exists* is not raised and the file is truncated to zero length.
- *no-fail* If the file already exists, the exception with condition type *&i/o-file-already-exists* is not raised, even if *no-create* is not included, and the file is truncated to zero length.
- *no-truncate* If the file already exists and the exception with condition type *&i/o-file-already-exists* has been inhibited by inclusion of *no-create* or *no-fail*, the file is not truncated, but the port's current position is still set to the beginning of the file.

3.13.3 Buffer modes

Macro: **buffer-mode** *buffer-mode-symbol*

[R6RS] *Buffer-mode-symbol* must be a symbol whose name is one of *none*, *line*, and *block*. The result is the corresponding symbol, and specifies the associated buffer mode.

Function: **buffer-mode?** *obj*

[R6RS] Returns #t if the argument is a valid buffer-mode symbol, and returns #f otherwise.

3.13.4 Transcoders

Function: **latin-1-codec**

Function: **utf-8-codec**
Function: **utf-16-codec**
[R6RS] These are predefined codecs for the ISO 8859-1, UTF-8, and UTF-16 encoding schemes.

Macro: **eol-style** *eol-style-symbol*
[R6RS] *Eol-style-symbol* should be a symbol whose name is one of *lf, cr, crlf, nel, crnel, ls,* and *none*. The form evaluates to the corresponding symbol. If the name of eol-style symbol is not one of these symbols, the effect and result are implementation-dependent; in particular, the result may be an eol-style symbol acceptable as an eol-style argument to make-transcoder. Otherwise, an exception is raised.

Function: **native-eol-style**
[R6RS] Returns the default end-of-line style of the underlying platform.

Condition Type: **&i/o-decoding**
Function: **make-i/o-decoding-error** *port*
Function: **i/o-decoding-error?** *obj*
[R6RS] An exception with this type is raised when one of the operations for textual input from a port encounters a sequence of bytes that cannot be translated into a character or string by the input direction of the port's transcoder.

When such an exception is raised, the port's position is past the invalid encoding.

Condition Type: **&i/o-encoding**
Function: **make-i/o-encoding-error** *port char*
Function: **i/o-encoding-error?** *obj*
Function: **i/o-encoding-error-char** *obj*
[R6RS] An exception with this type is raised when one of the operations for textual output to a port encounters a character that cannot be translated into bytes by the output direction of the port's transcoder. *Char* is the character that could not be encoded.

Macro: **error-handling-mode** *error-handling-mode-symbol*
[R6RS] *Error-handling-mode-symbol* should be a symbol whose name is one of *ignore, raise,* and *replace*. The form evaluates to the corresponding symbol.

Function: **make-transcoder** *codec :optional eol-style handling-mode*
[R6RS] *Codec* must be a codec; *eol-style*, if present, an eol-style symbol; and *handling-mode*, if present, an error-handling-mode symbol. *Eol-style* may be omitted, in which case it defaults to the native end-of-line style of the underlying platform. *Handling-mode* may be omitted, in which case it defaults to replace. The result is a transcoder with the behavior specified by its arguments.

Function: **native-transcoder**
[R6RS] Returns platform dependent transcoder.

Function: **transcoder-codec** *transcoder*
Function: **transcoder-eol-style** *transcoder*
Function: **transcoder-error-handling-mode** *transcoder*
[R6RS] These are accessors for transcoder objects; when applied to a transcoder returned by *make-transcoder*, they return the *codec, eol-style*, and *handling-mode* arguments, respectively.

Function: **bytevector->string** *bytevector transcoder*
[R6RS] Returns the string that results from transcoding the *bytevector* according to the input direction of the transcoder.

Function: **string->bytevector** *string transcoder*
[R6RS] Returns the bytevector that results from transcoding the *string* according to the output direction of the transcoder.

3.13.5 End-of-file object

Function: **eof-object**
[R6RS] Returns the end-of-file object.

Function: **eof-object?** *obj*
[R6RS] Returns #t if *obj* is the end-of-file object, #f otherwise.

3.13.6 Input and output ports

Function: **port?** *obj*
[R6RS] Returns #t if *obj* is port, #f otherwise.

Function: **port-transcoder** *port*
[R6RS] Returns the transcoder associated with port if port is textual and has an associated transcoder, and returns #f if port is binary or does not have an associated transcoder.

Function: **textual-port?** *obj*
Function: **binary-port?** *obj*
[R6RS] The *textual-port?* procedure returns #t if *port* is textual, and returns #f otherwise. The *binary-port?* procedure returns #t if *port* is binary, and returns #f otherwise.

Function: **transcoded-port** *binary-port transcoder*
[R6RS] The *transcoded-port* procedure returns a new textual port with the specified *transcoder*. Otherwise the new textual port's state is largely the same as that of *binary-port*. If *binary-port* is an input port, the new textual port will be an input port and will transcode the bytes that have not yet been read from *binary-port*. If *binary-port* is an output port, the new textual port will be an output port and will transcode output characters into bytes that are written to the byte sink represented by *binary-port*.

Function: **port-has-port-position?** *port*
Function: **port-position** *port*
[R6RS] The *port-has-port-position?* procedure returns #t if the port supports the *port-position* operation, and #f otherwise.

The *port-position* procedure returns the index of the position at which the next position would be read from or written to the port as an exact non-negative integer object.

Function: **port-has-set-port-position!?** *port*
Function: **set-port-position** *port pos*
[R6RS] The *port-has-set-port-position!?* procedure returns #t if the *port* supports the *set-port-position!* operation, and #f otherwise.

The *set-port-position!* procedure raises an exception with condition type *&assertion* if the *port* does not support the operation, and an exception with condition type *&i/o-invalid-position* if *pos* is not in the range of valid positions of port. Otherwise, it sets the current position of the *port* to *pos*. If *port* is an output port, *set-port-position!* first flushes port.

On Sagittarius Scheme, binary port and string port have port-position but transcoded file port does not support the *port-position* and *set-port-position!*.

Function: **close-port** *port*
[R6RS] Closes the port, rendering the port incapable of delivering or accepting data. If *port* is an output port, it is flushed before being closed. This has no effect if the port has already been closed. A closed port is still a port. The *close-port* procedure returns unspecified values.

Function: **call-with-port** *proc port*
[R6RS] *Proc* must accept one argument. The *call-with-port* procedure calls *proc* with *port* as an

argument. If *proc* returns, *port* is closed automatically and the values returned by *proc* are returned. If *proc* does not return, *port* is not closed automatically, except perhaps when it is possible to prove that port will never again be used for an input or output operation.

3.13.7 Input ports

`Function:` **input-port?** *obj*
[R6RS] Returns #t if *obj* is an input port (or a combined input and output port), #f otherwise.

`Function:` **port-eof?** *input-port*
[R6RS] Returns #t if the *lookahead-u8* procedure (if *input-port* is a binary port) or the *lookahead-char* procedure (if *input-port* is a textual port) would return the end-of-file object, and #f otherwise.

`Function:` **open-file-input-port** *filename :optional file-options buffer-mode maybe-transcoder*
[R6RS] *Maybe-transcoder* must be either a transcoder or #f.

The *file-options* argument, which may determine various aspects of the returned port, defaults to the value of (file-options).

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The *buffer-mode* argument defaults to *block*.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the returned port.

If *maybe-transcoder* is #f or absent, the port will be a binary port, otherwise the port will be textual port.

`Function:` **open-bytevector-input-port** *bytevector :optional maybe-transcoder*
[R6RS] *Maybe-transcoder* must be either a transcoder or #f.

The *open-bytevector-input-port* procedure returns an input port whose bytes are drawn from *bytevector*. If transcoder is specified, it becomes the transcoder associated with the returned port.

If *maybe-transcoder* is #f or absent, the port will be a binary port, otherwise the port will be textual port.

`Function:` **open-string-input-port** *string*
[R6RS] Returns a textual input port whose characters are drawn from *string*.

These procedures reuse the given arguments, thus if *bytevector* is modified after *open-bytevector-input-port* has been called, it affects the result of the port. So does *open-string-input-port*.

`Function:` **standard-input-port**
[R6RS] Returns a fresh binary input port connected to standard input.

`Function:` **current-input-port** *:optional port*
[R6RS+] If *port* is given, the *current-input-port* sets *port* as a default port for input. Otherwise it returns a default input port.

`Function:` **make-custom-binary-input-port** *id read! get-position set-position! close*
[R6RS] Returns a newly created binary input port whose byte source is an arbitrary algorithm represented by the *read!* procedure. *Id* must be a string naming the new port, provided for informational purposes only. *Read!* must be a procedure and should behave as specified below; it will be called by operations that perform binary input.

Each of the remaining arguments may be #f; if any of those arguments is not #f, it must be a procedure and should behave as specified below.

- (*read! bytevector start count*)

Start will be a non-negative exact integer object, count will be a positive exact integer object, and *bytevector* will be a bytevector whose length is at least *start* + *count*. The *read!* procedure should obtain up to count bytes from the byte source, and should write those bytes into *bytevector* starting at index start. The *read!* procedure should return an exact integer object. This integer object should represent the number of bytes that it has read. To indicate an end of file, the *read!* procedure should write no bytes and return 0.

- (*get-position*)

  The *get-position* procedure (if supplied) should return an exact integer object representing the current position of the input port. If not supplied, the custom port will not support the *port-position* operation.

- (*set-position! pos*)

  Pos will be a non-negative exact integer object. The *set-position!* procedure (if supplied) should set the position of the input port to *pos*. If not supplied, the custom port will not support the *set-port-position!* operation.

- (*close*)

  The *close* procedure (if supplied) should perform any actions that are necessary when the input port is closed.

`Function:` **make-custom-textual-input-port** *id read! get-position set-position! close*

[R6RS] Returns a newly created textual input port whose character source is an arbitrary algorithm represented by the *read!* procedure. *Id* must be a string naming the new port, provided for informational purposes only. *Read!* must be a procedure and should behave as specified below; it will be called by operations that perform textual input.

Each of the remaining arguments may be #f; if any of those arguments is not #f, it must be a procedure and should behave as specified below.

- (*read! string start count*)

  Start will be a non-negative exact integer object, count will be a positive exact integer object, and *string* will be a string whose length is at least *start* + *count*. The *read!* procedure should obtain up to count characters from the character source, and should write those characters into string starting at index start. The *read!* procedure should return an exact integer object representing the number of characters that it has written. To indicate an end of file, the read! procedure should write no bytes and return 0.

- (*get-position*)

  The *get-position* procedure (if supplied) should return a single value. The return value should represent the current position of the input port. If not supplied, the custom port will not support the *port-position* operation.

- (*set-position! pos*)

  The *set-position!* procedure (if supplied) should set the position of the input port to *pos* if *pos* is the return value of a call to *get-position*. If not supplied, the custom port will not support the *set-port-position!* operation.

- (*close*)

  The *close* procedure (if supplied) should perform any actions that are necessary when the input port is closed.

3.13.8 Binary input

Function:**get-u8** *binary-input-port*

[R6RS] Reads from *binary-input-port*, blocking as necessary, until a byte is available from *binary-input-port* or until an end of file is reached. If a byte becomes available, *get-u8* returns the byte as an octet and updates *binary-input-port* to point just past that byte. If no input byte is seen before an end of file is reached, the end-of-file object is returned.

Function:**lookahead-u8** *binary-input-port*

[R6RS] The *lookahead-u8* procedure is like *get-u8*, but it does not update *binary-input-port* to point past the byte.

Function:**get-bytevector-n** *binary-input-port count :optional reckless*

[R6RS] *Count* must be an exact, non-negative integer object representing the number of bytes to be read.

The *get-bytevector-n* procedure reads from *binary-input-port*, blocking as necessary, until *count* bytes are available from *binary-input-port* or until an end of file is reached. If *count* bytes are available before an end of file, *get-bytevector-n* returns a bytevector of size *count*. If fewer bytes are available before an end of file, *get-bytevector-n* returns a bytevector containing those bytes. In either case, the input port is updated to point just past the bytes read. If an end of file is reached before any bytes are available, *get-bytevector-n* returns the end-of-file object.

Function:**get-bytevector-n!** *binary-input-port bytevector start count :optional reckless*

[R6RS] *Count* must be an exact, non-negative integer object, representing the number of bytes to be read. *bytevector* must be a bytevector with at least *start* + *count* elements.

The *get-bytevector-n!* procedure reads from *binary-input-port,* blocking as necessary, until *count* bytes are available from *binary-input-port* or until an end of file is reached. If *count* bytes are available before an end of file, they are written into *bytevector* starting at index *start*, and the result is *count*. If fewer bytes are available before the next end of file, the available bytes are written into *bytevector* starting at index *start*, and the result is a number object representing the number of bytes actually read. In either case, the input port is updated to point just past the bytes read. If an end of file is reached before any bytes are available, *get-bytevector-n!* returns the end-of-file object.

Function:**get-bytevector-some** *binary-input-port :optional reckless*

[R6RS] Reads from *binary-input-port*, blocking as necessary, until bytes are available from *binary-input-port* or until an end of file is reached. If bytes become available, *get-bytevector-some* returns a freshly allocated bytevector containing the initial available bytes (at least one and maximum 512 bytes), and it updates *binary-input-port* to point just past these bytes. If no input bytes are seen before an end of file is reached, the end-of-file object is returned.

Function:**get-bytevector-all** *binary-input-port :optional reckless*

[R6RS] Attempts to read all bytes until the next end of file, blocking as necessary. If one or more bytes are read, *get-bytevector-all* returns a bytevector containing all bytes up to the next end of file. Otherwise, *get-bytevector-all* returns the end-of-file object.

These procedures can take optional argument *reckless*. If this is given, these procedures can read bytes from textual port. This optional argument is for socket programming. Users needs to make sure that the given port can be read as textual port after reading port recklessly.

3.13.9 Textual Input

Function:**get-char** *textual-input-port*

[R6RS] Reads from *textual-input-port*, blocking as necessary, until a complete character is available from textual-input-port, or until an end of file is reached.

If a complete character is available before the next end of file, *get-char* returns that character and

updates the input port to point past the character. If an end of file is reached before any character is read, *get-char* returns the end-of-file object.

`Function:` **lookahead-char** *textual-input-port*

[R6RS] The *lookahead-char* procedure is like *get-char*, but it does not update *textual-input-port* to point past the character.

`Function:` **get-string-n** *textual-input-port count*

[R6RS] *Count* must be an exact, non-negative integer object, representing the number of characters to be read.

The *get-string-n* procedure reads from textual-input-port, blocking as necessary, until *count* characters are available, or until an end of file is reached.

If *count* characters are available before end of file, *get-string-n* returns a string consisting of those *count* characters. If fewer characters are available before an end of file, but one or more characters can be read, *get-string-n* returns a string containing those characters. In either case, the input port is updated to point just past the characters read. If no characters can be read before an end of file, the end-of-file object is returned.

`Function:` **get-string-n!** *textual-input-port string start count*

[R6RS] *Start* and *count* must be exact, non-negative integer objects, with *count* representing the number of characters to be read. *String* must be a string with at least *start* + *count* characters.

The *get-string-n!* procedure reads from *textual-input-port* in the same manner as *get-string-n*. If *count* characters are available before an end of file, they are written into *string* starting at index *start*, and *count* is returned. If fewer characters are available before an end of file, but one or more can be read, those characters are written into *string* starting at index *start* and the number of characters actually read is returned as an exact integer object. If no characters can be read before an end of file, the end-of-file object is returned.

`Function:` **get-string-all** *textual-input-port*

[R6RS] Reads from *textual-input-port* until an end of file, decoding characters in the same manner as *get-string-n* and *get-string-n!*.

If characters are available before the end of file, a string containing all the characters decoded from that data are returned. If no character precedes the end of file, the end-of-file object is returned.

`Function:` **get-line** *textual-input-port*

[R6RS] Reads from *textual-input-port* up to and including the linefeed character or end of file, decoding characters in the same manner as *get-string-n* and *get-string-n!*.

If a linefeed character is read, a string containing all of the text up to (but not including) the linefeed character is returned, and the port is updated to point just past the linefeed character. If an end of file is encountered before any linefeed character is read, but some characters have been read and decoded as characters, a string containing those characters is returned. If an end of file is encountered before any characters are read, the end-of-file object is returned.

`Function:` **get-datum** *textual-input-port*

[R6RS] Reads an external representation from *textual-input-port* and returns the datum it represents. The *get-datum* procedure returns the next datum that can be parsed from the given *textual-input-port*, updating *textual-input-port* to point exactly past the end of the external representation of the object.

If a character inconsistent with an external representation is encountered in the input, an exception with condition types *&lexical* and *&i/o-read* is raised. Also, if the end of file is encountered after the beginning of an external representation, but the external representation is incomplete and therefore cannot be parsed, an exception with condition types *&lexical* and *&i/o-*

read is raised.

3.13.10 Output ports

Function: **output-port?** *obj*

[R6RS] Returns #t if *obj* is an output port (or a combined input and output port), #f otherwise.

Function: **flush-output-port** *output-port*

[R6RS] Flushes any buffered output from the buffer of *output-port* to the underlying file, device, or object. The *flush-output-port* procedure returns unspecified values.

Function: **output-port-buffer-mode** *output-port*

[R6RS] Returns the symbol that represents the buffer mode of *output-port*.

Function: **open-file-output-port** *filename :optional file-options buffer-mode maybe-transcoder*

[R6RS] *Maybe-transcoder* must be either a transcoder or #f.

The *open-file-output-port* procedure returns an output port for the named file.

The *file-options* argument, which may determine various aspects of the returned port, defaults to the value of (file-options).

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The buffer-mode argument defaults to *block*.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the port.

If *maybe-transcoder* is #f or absent, the port will be a binary port, otherwise the port will be textual port.

Function: **open-bytevector-output-port** *:optional maybe-transcoder*

[R6RS] *Maybe-transcoder* must be either a transcoder or #f.

The *open-bytevector-output-port* procedure returns two values: an output port and an extraction procedure. The output port accumulates the bytes written to it for later extraction by the procedure.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the port. If *maybe-transcoder* is #f or absent, the port will be a binary port, otherwise the port will be textual port.

The extraction procedure takes no arguments. When called, it returns a bytevector consisting of all the port's accumulated bytes (regardless of the port's current position), removes the accumulated bytes from the port, and resets the port's position.

Function: **call-with-bytevector-output-port** *proc :optional maybe-transcoder*

[R6RS] *Proc* must accept one argument. *Maybe-transcoder* must be either a transcoder or #f.

The *call-with-bytevector-output-port* procedure creates an output port that accumulates the bytes written to it and calls *proc* with that output port as an argument. Whenever *proc* returns, a bytevector consisting of all of the port's accumulated bytes (regardless of the port's current position) is returned and the port is closed.

The transcoder associated with the output port is determined as for a call to *open-bytevector-output-port*.

Function: **open-string-output-port**

[R6RS] Returns two values: a textual output port and an extraction procedure. The output port accumulates the characters written to it for later extraction by the procedure.

The extraction procedure takes no arguments. When called, it returns a string consisting of all of the port's accumulated characters (regardless of the current position), removes the accumulated characters from the port, and resets the port's position.

Function: **call-with-string-output-port** *proc*

[R6RS] *Proc* must accept one argument. The *call-with-string-output-port* procedure creates a textual output port that accumulates the characters written to it and calls *proc* with that output port as an argument. Whenever *proc* returns, a string consisting of all of the port's accumulated characters (regardless of the port's current position) is returned and the port is closed.

Function: **standard-output-port**
Function: **standard-error-port**

[R6RS] Returns a fresh binary output port connected to the standard output or standard error respectively.

Function: **current-output-port** *:optional port*
Function: **current-error-port** *:optional port*

[R6RS+] If *port* is given, these procedures set *port* as a default port for output and error. These return default ports for regular output and error output.

Function: **make-custom-binary-output-port** *id write! get-position set-position! close*

[R6RS] Returns a newly created binary output port whose byte sink is an arbitrary algorithm represented by the *write!* procedure. *Id* must be a string naming the new port, provided for informational purposes only. *Write!* must be a procedure and should behave as specified below; it will be called by operations that perform binary output.

Each of the remaining arguments may be #f; if any of those arguments is not #f, it must be a procedure and should behave as specified in the description of *make-custom-binary-input-port*.

- (*write! bytevector start count*)

  Start and count will be non-negative exact integer objects, and *bytevector* will be a bytevector whose length is at least *start* + *count*. The *write!* procedure should write up to *count* bytes from *bytevector* starting at index *start* to the byte sink. If count is 0, the *write!* procedure should have the effect of passing an end-of-file object to the byte sink. In any case, the *write!* procedure should return the number of bytes that it wrote, as an exact integer object.

Function: **make-custom-textual-output-port** *id write! get-position set-position! close*

[R6RS] Returns a newly created textual output port whose byte sink is an arbitrary algorithm represented by the *write!* procedure. *Id* must be a string naming the new port, provided for informational purposes only. *Write!* must be a procedure and should behave as specified below; it will be called by operations that perform textual output.

Each of the remaining arguments may be #f; if any of those arguments is not #f, it must be a procedure and should behave as specified in the description of *make-custom-textual-input-port*.

- (*write! string start count*)

  Start and *count* will be non-negative exact integer objects, and *string* will be a string whose length is at least *start* + *count*. The *write!* procedure should write up to *count* characters from *string* starting at index *start* to the character sink. If count is 0, the *write!* procedure should have the effect of passing an end-of-file object to the character sink. In any case, the *write!* procedure should return the number of characters that it wrote, as an exact integer object.

3.13.11 Binary output

Function: **put-u8** *binary-output-port octet*

[R6RS] Writes *octet* to the output port and returns unspecified values.

Function: **put-bytevector** *binary-output-port bytevector :optional start count*

[R6RS] *Start* and *count* must be non-negative exact integer objects that default to 0 and (bytevector-length *bytevector*) - *start*, respectively. *Bytevector* must have a length of at least *start* + *count*. The *put-bytevector* procedure writes the *count* bytes of the bytevector *bytevector* starting at index *start* to the output port. The *put-bytevector* procedure returns unspecified values.

3.13.12 Textual output

Function:**put-char** *textual-output-port char*

[R6RS] Writes *char* to the port and returns unspecified values.

Function:**put-string** *textual-output-port string :optional start count*

[R6RS] *Start* and *count* must be non-negative exact integer objects. *String* must have a length of at least *start* + *count*. *Start* defaults to 0. Count defaults to (string-length *string*) - *start*. The *put-string* procedure writes the *count* characters of *string* starting at index *start* to the port. The *put-string* procedure returns unspecified values.

Function:**put-datum** *textual-output-port datum*

[R6RS] *Datum* should be a datum value. The *put-datum* procedure writes an external representation of datum to *textual-output-port*.

3.13.13 Input/output ports

Function:**open-file-input/output-port** *filename :optional file-options buffer-mode transcoder*

[R6RS] Returns a single port that is both an input port and an output port for the named file. The optional arguments default as described in the specification of *open-file-output-port*. If the input/output port supports *port-position* and/or *set-port-position!*, the same port position is used for both input and output.

Function:**make-custom-binary-input/output-port** *id read! write! get-position set-position! close*

[R6RS] Returns a newly created binary input/output port whose byte source and sink are arbitrary algorithms represented by the *read!* and *write!* procedures. *Id* must be a string naming the new port, provided for informational purposes only. *Read!* and *write!* must be procedures, and should behave as specified for the *make-custom-binary-input-port* and *make-custom-binary-output-port* procedures.

Each of the remaining arguments may be #f; if any of those arguments is not #f, it must be a procedure and should behave as specified in the description of *make-custom-binary-input-port*.

Function:**make-custom-textual-input/output-port** *id read! write! get-position set-position! close*

[R6RS] Returns a newly created textual input/output port whose textual source and sink are arbitrary algorithms represented by the *read!* and *write!* procedures. *Id* must be a string naming the new port, provided for informational purposes only. *Read!* and *write!* must be procedures, and should behave as specified for the *make-custom-textual-input-port* and *make-custom-textual-output-port* procedures.

Each of the remaining arguments may be #f; if any of those arguments is not #f, it must be a procedure and should behave as specified in the description of *make-custom-textual-input-port*.

3.13.14 Simple I/O

This section describes the *(rnrs io simple (6))*library, which provides a somewhat more convenient interface for performing textual I/O on ports. This library also exports the same procedures as *(rnrs io posts (6))*library. I do not write the documentation of it, if you want to import only this library, make sure which procedures are exported. You can see it on R6RS.

Function:**call-with-input-file** *filename proc*
Function:**call-with-output-file** *filename proc*

[R6RS] *Proc* should accept one argument. These procedures open the file named by *filename* for input or for output, with no specified file options, and call *proc* with the obtained port as an argument. If *proc* returns, the port is closed automatically and the values returned by *proc* are returned. If *proc* does not return, the port is not closed automatically, unless it is possible to prove that the port will never again be used for an I/O operation.

Function:**with-input-from-file** *filename thunk*
Function:**with-output-to-file** *filename thunk*

[R6RS] *Thunk* must be a procedure and must accept zero arguments. The file is opened for input or output using empty file options, and *thunk* is called with no arguments. These procedure replace current input/output port during *thunk* is being called. When *thunk* returns, the port is closed automatically. The values returned by *thunk* are returned.

Function:**open-input-file** *filename*
Function:**open-output-file** *filename*

[R6RS] Opens *filename* for input/output, with empty file options, and returns the obtained port.

Function:**close-input-port** *port*
Function:**close-output-port** *port*

[R6RS] Closes *input-port* or *output-port*, respectively.

Function:**read-char** *:optional textual-input-port*
Function:**peek-char** *:optional textual-input-port*

[R6RS] These work the same as *get-char* and *lookahead-char.*

If *textual-input-port* is omitted, it defaults to the value returned by current-input-port.

Function:**read** *:optional textual-input-port*

[R6RS] Reads an external representation from *textual-input-port* and returns the datum it represents. The read procedure operates in the same way as *get-datum*.

If *textual-input-port* is omitted, it defaults to the value returned by *current-input-port*.

Function:**write-char** *char :optional textual-output-port*

[R6RS] Writes an encoding of the character char to the *textual-output-port*, and returns unspecified values.

If *textual-output-port* is omitted, it defaults to the value returned by *current-output-port*.

Function:**newline** *:optional textual-output-port*

[R6RS] This is equivalent to using *write-char* to write *#\linefeed* to *textual-output-port*.

If *textual-output-port* is omitted, it defaults to the value returned by *current-output-port*.

Function:**display** *obj :optional textual-output-port*

[R6RS] Writes a representation of *obj* to the given *textual-output-port*. Strings that appear in the written representation are not enclosed in double quotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by *write-char* instead of by *write*. The *display* procedure returns unspecified values.

If *textual-output-port* is omitted, it defaults to the value returned by *current-output-port*.

Function:**write** *obj :optional textual-output-port*

[R6RS] Writes the external representation of *obj* to *textual-output-port*. The *write* procedure operates in the same way as *put-datum*.

If *textual-output-port* is omitted, it defaults to the value returned by *current-output-port*.

3.14 (rnrs files (6))

Function:**file-exists?** *filename*

[R6RS] *Filename* must be string. The *file-exists?* procedure returns #t if the named file exists at the time the procedure is called, #f otherwise.

Function:**delete-file** *filename*

[R6RS] *Filename* must be a file name. The *delete-file* procedure deletes the named file if it exists and can be deleted, and returns unspecified values. If the file does not exist or cannot be deleted, an exception with condition type *&i/o-filename* is raised.

3.15 (rnrs programs (6))

Function:**command-line**

[R6RS] Returns a nonempty list of strings. The first element is the running script file name or '() on REPL. The remaining elements are command-line arguments according to the operating system's conventions.

Function:**exit** *:optional obj*

[R6RS] Exits the running program and communicates an exit value to the operating system. If no argument is supplied, the *exit* procedure should communicate to the operating system that the program exited normally. If an argument is supplied and if it is fixnum, the *exit* procedure translates it into an appropriate exit value for the operating system. Otherwise the exit is assumed to be abnormal.

3.16 Arithmetic libraries

This chapter describes *(rnrs arithmetic bitwise (6)), (rnrs arithmetic fixnum (6))* and *(rnrs arithmetic flonum (6))* libraries.

3.16.1 Fixnum

On Sagittarius Scheme, fixnum is 30 bits or 62 bits depending on platform. More generically, it will be calculated like $2^{(8n-2)}$ as max fixnum size, *n* is platform pointer size. ex) On 32 bits platform it is 4. However, non 32 bits platform is not well tested so if you find a bug please send a report.

This section uses *fx, fx1 fx2,* etc., as parameter names for arguments that must be fixnums.

Function:**fixnum?** *obj*

[R6RS] Returns #t if *obj* is an exact integer object within the fixnum range, #f otherwise.

Function:**fixnum-width**
Function:**least-fixnum**
Function:**greatest-fixnum**

[R6RS] These procedures returns bit size of fixnum, minimum and maximum value of the fixnum range.

Function:**fx=?** *fx1 fx2 fx3 ...*
Function:**fx>?** *fx1 fx2 fx3 ...*
Function:**fx<?** *fx1 fx2 fx3 ...*
Function:**fx>=?** *fx1 fx2 fx3 ...*
Function:**fx<=?** *fx1 fx2 fx3 ...*

[R6RS] These procedures return #t if their arguments are: equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, #f otherwise.

Function: **fxzero?** *fx*
Function: **fxpositive?** *fx*
Function: **fxnegative?** *fx*
Function: **fxodd?** *fx*
Function: **fxeven?** *fx*

[R6RS] These numerical predicates test a fixnum for a particular property, returning #t or #f. The five properties tested by these procedures are: whether the number object is zero, greater than zero, less than zero, odd, or even.

Function: **fxmax** *fx1 fx2 ...*
Function: **fxmin** *fx1 fx2 ...*

[R6RS] These procedures return the maximum or minimum of their arguments.

Function: **fx+** *fx1 fx2*
Function: **fx*** *fx1 fx2*

[R6RS] These procedures return the sum or product of their arguments, provided that sum or product is a fixnum. An exception with condition type *&implementation-restriction* is raised if that sum or product is not a fixnum.

Function: **fx-** *fx1 :optional fx2*

[R6RS] With two arguments, this procedure returns the difference *fx1 - fx2*, provided that difference is a fixnum.

With one argument, this procedure returns the additive inverse of its argument, provided that integer object is a fixnum.

An exception with condition type *&implementation-restriction* is raised if the mathematically correct result of this procedure is not a fixnum.

NOTE: R6RS says it raises *&assertion* if the result is not fixnum, however Sagittarius raises *&implementation-restriction* for consistency with *fx+* and *fx**.

Function: **fxdiv-and-mode** *fx1 fx2*
Function: **fxdiv** *fx1 fx2*
Function: **fxmod** *fx1 fx2*
Function: **fxdiv0-and-mod0** *fx1 fx2*
Function: **fxdiv0** *fx1 fx2*
Function: **fxmod0** *fx1 fx2*

[R6RS] *Fx2* must be nonzero. These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in *(rnrs base (6))* section.

Function: **fx+/carry** *fx1 fx2 fx3*

[R6RS] Returns the two fixnum results of the following computation:

```
(let* ((s (+ fx1 fx2 fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

Function: **fx-/carry** *fx1 fx2 fx3*

[R6RS] Returns the two fixnum results of the following computation:

```
(let* ((d (- fx1 fx2 fx3))
       (d0 (mod0 d (expt 2 (fixnum-width))))
```

(d1 (div0 d (expt 2 (fixnum-width)))))

(values d0 d1))

Function: **fx*/carry** *fx1 fx2 fx3*

[R6RS] Returns the two fixnum results of the following computation:

(let ((s (+ (* fx1 fx2) fx3))*

(s0 (mod0 s (expt 2 (fixnum-width))))

(s1 (div0 s (expt 2 (fixnum-width)))))

(values s0 s1))

Function: **fxnot** *fx*

[R6RS] Returns the unique fixnum that is congruent mod $2^{(8n-2)}$ to the one's-complement of *fx*.

Function: **fxand** *fx1 ...*
Function: **fxior** *fx1 ...*
Function: **fxxor** *fx1 ...*

[R6RS] These procedures return the fixnum that is the bit-wise "and", "inclusive or", or "exclusive or" of the two's complement representations of their arguments. If they are passed only one argument, they return that argument. If they are passed no arguments, they return the fixnum (either - 1 or 0) that acts as identity for the operation.

Function: **fxif** *fx1 fx2 fx3*

[R6RS] Returns the fixnum that is the bit-wise "if" of the two's complement representations of its arguments, i.e. for each bit, if it is 1 in *fx1*, the corresponding bit in *fx2* becomes the value of the corresponding bit in the result, and if it is 0, the corresponding bit in *fx3* becomes the corresponding bit in the value of the result. This is the fixnum result of the following computation:

(fxior (fxand fx1 fx2)

(fxand (fxnot fx1) fx3))

Function: **fxbit-count** *fx*

[R6RS] If *fx* is non-negative, this procedure returns the number of 1 bits in the two's complement representation of *fx*. Otherwise it returns the result of the following computation:

(fxnot (fxbit-count (fxnot ei)))

Function: **fxlength** *fx*

[R6RS] Returns the number of bits needed to represent *fx* if it is positive, and the number of bits needed to represent *(fxnot fx)* if it is negative, which is the fixnum result of the following computation:

(do ((result 0 (+ result 1))

(bits (if (fxnegative? fx)

(fxnot fx)

fx)

(fxarithmetic-shift-right bits 1)))

((fxzero? bits)

result))

Function: **fxfirst-bit-set** *fx*

[R6RS] Returns the index of the least significant 1 bit in the two's complement representation of

fx. If *fx* is 0, then - 1 is returned.

Function: **fxfbit-set?** *fx1 fx2*

[R6RS] *Fx2* must be non-negative and less than *(fixnum-width)*. The *fxbit-set?* procedure returns #t if the *fx2*th bit is 1 in the two's complement representation of *fx1*, and #f otherwise. This is the fixnum result of the following computation:

```
(not
 (fxzero?
  (fxand fx1
         (fxarithmetic-shift-left 1 fx2))))
```

Function: **fxcopy-bit** *fx1 fx2 fx3*

[R6RS] *Fx2* must be non-negative and less than *(fixnum-width)*. *Fx3* must be 0 or 1. The *fxcopy-bit* procedure returns the result of replacing the *fx2*th bit of *fx1* by *fx3*, which is the result of the following computation:

```
(let* ((mask (fxarithmetic-shift-left 1 fx2)))
 (fxif mask
       (fxarithmetic-shift-left fx3 fx2)
       fx1))
```

Function: **fxbit-field** *fx1 fx2 fx3*

[R6RS] *Fx2* and *fx3* must be non-negative and less than *(fixnum-width)*. Moreover, *fx2* must be less than or equal to *fx3*. The *fxbit-field* procedure returns the number represented by the bits at the positions from *fx2* (inclusive) to *fx3* (exclusive), which is the fixnum result of the following computation:

```
(let* ((mask (fxnot
              (fxarithmetic-shift-left -1 fx3))))
 (fxarithmetic-shift-right (fxand fx1 mask)
                           fx2))
```

Function: **fxcopy-bit-field** *fx1 fx2 fx3 fx4*

[R6RS] *Fx2* and *fx3* must be non-negative and less than *(fixnum-width)*. Moreover, *fx2* must be less than or equal to *fx3*. The *fxcopy-bit-field* procedure returns the result of replacing in *fx1* the bits at positions from *fx2* (inclusive) to *fx3* (exclusive) by the corresponding bits in *fx4*, which is the fixnum result of the following computation:

```
(let* ((to    fx1)
       (start fx2)
       (end   fx3)
       (from  fx4)
       (mask1 (fxarithmetic-shift-left -1 start))
       (mask2 (fxnot
               (fxarithmetic-shift-left -1 end)))
       (mask (fxand mask1 mask2)))
 (fxif mask
       (fxarithmetic-shift-left from start)
```

to))

Function: **fxarithmetic-shift** *fx1 fx2*

[R6RS] The absolute value of *fx2* must be less than *(fixnum-width)*. If

(floor (fx1 (expt 2 fx2)))*

is a fixnum, then that fixnum is returned. Otherwise an exception with condition type *&implementation-restriction* is raised.

Function: **fxarithmetic-shift-left** *fx1 fx2*
Function: **fxarithmetic-shift-right** *fx1 fx2*

[R6RS] *Fx2* must be non-negative, and less than *(fixnum-width)*. The *fxarithmetic-shift-left* procedure behaves the same as *fxarithmetic-shift*, and *(fxarithmetic-shift-right fx1 fx2)* behaves the same as *(fxarithmetic-shift fx1 (fx- fx2))*.

Function: **fxrotate-bit-field** *fx1 fx2 fx3 fx4*

[R6RS] *Fx2*, *fx3*, and *fx4* must be non-negative and less than *(fixnum-width)*. *Fx2* must be less than or equal to *fx3*. *Fx4* must be less than the difference between *fx3* and *fx2*. The *fxrotate-bit-field* procedure returns the result of cyclically permuting in *fx1* the bits at positions from *fx2* (inclusive) to *fx3* (exclusive) by *fx4* bits towards the more significant bits, which is the result of the following computation:

```
(let* ((n     fx1)
       (start fx2)
       (end   fx3)
       (count fx4)
       (width (fx- end start)))
  (if (fxpositive? width)
      (let* ((count (fxmod count width))
             (field0
               (fxbit-field n start end))
             (field1
               (fxarithmetic-shift-left
                 field0 count))
             (field2
               (fxarithmetic-shift-right
                 field0 (fx- width count)))
             (field (fxior field1 field2)))
        (fxcopy-bit-field n start end field))
      n))
```

Function: **fxreverse-bit-field** *fx1 fx2 fx3*

[R6RS] *Fx2* and *fx3* must be non-negative and less than *(fixnum-width)*. Moreover, *fx2* must be less than or equal to *fx3*. The *fxreverse-bit-field* procedure returns the fixnum obtained from *fx1* by reversing the order of the bits at positions from *fx2* (inclusive) to *fx3* (exclusive).

3.16.2 Flonums

This section uses *fl, fl1, fl2*, etc., as parameter names for arguments that must be flonums, and *ifl* as a name for arguments that must be integer-valued flonums, i.e., flonums for which the *integer-valued?* predicate returns true.

`Function:` **flonum?** *obj*
[R6RS] Returns #t if obj is a flonum, #f otherwise.

`Function:` **real->flonum** *x*
[R6RS] Returns the best flonum representation of *x*.

The value returned is a flonum that is numerically closest to the argument.

`Function:` **fl=?** *fl1 fl2 fl3 ...*
`Function:` **fl>?** *fl1 fl2 fl3 ...*
`Function:` **fl<?** *fl1 fl2 fl3 ...*
`Function:` **fl>=?** *fl1 fl2 fl3 ...*
`Function:` **fl<=?** *fl1 fl2 fl3 ...*
[R6RS] These procedures return #t if their arguments are: equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, #f otherwise. These predicates must be transitive.

`Function:` **flinteger?** *fl*
`Function:` **flzero?** *fl*
`Function:` **flpositive?** *fl*
`Function:` **flnegative?** *fl*
`Function:` **flodd?** *fl*
`Function:` **fleven?** *fl*
`Function:` **flfinite?** *fl*
`Function:` **flinfinite?** *fl*
`Function:` **flnan?** *fl*
[R6RS] These numerical predicates test a flonum for a particular property, returning #t or #f. The *flinteger?* procedure tests whether the number object is an integer, *flzero?* tests whether it is *fl=?* to zero, *flpositive?* tests whether it is greater than zero, *flnegative?* tests whether it is less than zero, *flodd?* tests whether it is odd, *fleven?* tests whether it is even, *flfinite?* tests whether it is not an infinity and not a NaN, *flinfinite?* tests whether it is an infinity, and *flnan?* tests whether it is a NaN.

`Function:` **flmax** *fl1 fl2 ...*
`Function:` **flmin** *fl1 fl2 ...*
[R6RS] These procedures return the maximum or minimum of their arguments. They always return a NaN when one or more of the arguments is a NaN.

`Function:` **fl+** *fl1 ...*
`Function:` **fl*** *fl1 ...*
[R6RS] These procedures return the flonum sum or product of their flonum arguments.

`Function:` **fl-** *fl1 fl2...*
`Function:` **fl-** *fl1*
`Function:` **fl/** *fl1 fl2...*
`Function:` **fl/** *fl1*
[R6RS] With two or more arguments, these procedures return the flonum difference or quotient of their flonum arguments, associating to the left. With one argument, however, they return the additive or multiplicative flonum inverse of their argument.

`Function:` **flabs** *fl*

[R6RS] Returns the absolute value of *fl*.

Function: **fldiv-and-mod** *fl1 fl2*
Function: **fldiv** *fl1 fl2*
Function: **flmod** *fl1 fl2*
Function: **fldiv0-and-mod0** *fl1 fl2*
Function: **fldiv0** *fl1 fl2*
Function: **flmod0** *fl1 fl2*

[R6RS] These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in report section on “Integer division”. For zero divisors, these procedures may return a NaN or some unspecified flonum.

Function: **flnumerator** *fl*
Function: **fldenominator** *fl*

[R6RS] These procedures return the numerator or denominator of *fl* as a flonum; the result is computed as if *fl* was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0.0 is defined to be 1.0.

Function: **flfloor** *fl*
Function: **flceiling** *fl*
Function: **fltruncate** *fl*
Function: **flround** *fl*

[R6RS] These procedures return integral flonums for flonum arguments that are not infinities or NaNs. For such arguments, *flfloor* returns the largest integral flonum not larger than *fl*. The *flceiling* procedure returns the smallest integral flonum not smaller than *fl*. The *fltruncate* procedure returns the integral flonum closest to *fl* whose absolute value is not larger than the absolute value of *fl*. The *flround* procedure returns the closest integral flonum to *fl*, rounding to even when *fl* represents a number halfway between two integers.

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

Function: **flexp** *fl*
Function: **fllog** *fl1 :optional fl2*
Function: **flsin** *fl*
Function: **flcos** *fl*
Function: **fltan** *fl*
Function: **flasin** *fl*
Function: **flacos** *fl*
Function: **flatan** *fl1 :optional fl2*

[R6RS] These procedures compute the usual transcendental functions. The *flexp* procedure computes the base-e exponential of *fl*. The *fllog* procedure with a single argument computes the natural logarithm of *fl1* (not the base ten logarithm); (*fllog fl1 fl2*) computes the base-*fl2* logarithm of *fl1*. The *flasin*, *flacos*, and *flatan* procedures compute arcsine, arccosine, and arctangent, respectively. (*flatan fl1 fl2*) computes the arc tangent of *fl1/fl2*.

Function: **flsqrt** *fl*

[R6RS] Returns the principal square root of *fl*.

Function: **flexpt** *fl1 fl2*

[R6RS] Either *fl1* should be non-negative, or, if *fl1* is negative, *fl2* should be an integer object. The *flexpt* procedure returns *fl1* raised to the power *fl2*. If fl1 is negative and *fl2* is not an integer object, the result may be a NaN, or may be some unspecified flonum. If *fl1* is zero, then the result is zero.

Condition Type: **&no-infinities**
Function: **make-no-infinities-violation** *obj*
Function: **no-infinities-violation?** *Obj*
Condition Type: **&no-nan**
Function: **make-no-nan-violation** *obj*
Function: **no-nan-violation?** *obj*

[R6RS] These types describe that a program has executed an arithmetic operations that is specified to return an infinity or a NaN, respectively.

Function: **fixnum->flonum** *fx*

[R6RS] Returns a flonum that is numerically closest to *fx*.

3.16.3 Exact bitwise arithmetic

This section uses ei, ei1, ei2, etc., as parameter names that must be exact integer objects.

Function: **bitwise-not** *ei*

[R6RS] Returns the exact integer object whose two's complement representation is the one's complement of the two's complement representation of *ei*.

Function: **bitwise-and** *ei1 ...*
Function: **bitwise-ior** *ei1 ...*
Function: **bitwise-xor** *ei1 ...*

[R6RS] These procedures return the exact integer object that is the bit-wise "and", "inclusive or", or "exclusive or" of the two's complement representations of their arguments. If they are passed only one argument, they return that argument. If they are passed no arguments, they return the integer object (either - 1 or 0) that acts as identity for the operation.

Function: **bitwise-if** *ei1 ei2 ei3*

[R6RS] Returns the exact integer object that is the bit-wise "if" of the two's complement representations of its arguments, i.e. for each bit, if it is 1 in *ei1*, the corresponding bit in *ei2* becomes the value of the corresponding bit in the result, and if it is 0, the corresponding bit in *ei3* becomes the corresponding bit in the value of the result. This is the result of the following computation:

```
(bitwise-ior (bitwise-and ei1 ei2)
             (bitwise-and (bitwise-not ei1) ei3))
```

Function: **bitwise-bit-count** *ei*

[R6RS] If *ei* is non-negative, this procedure returns the number of 1 bits in the two's complement representation of *ei*. Otherwise it returns the result of the following computation:

```
(bitwise-not (bitwise-bit-count (bitwise-not ei)))
```

Function: **bitwise-bit-length** *ei*

[R6RS] Returns the number of bits needed to represent *ei* if it is positive, and the number of bits needed to represent *(bitwise-not ei)* if it is negative, which is the exact integer object that is the result of the following computation:

```
(do ((result 0 (+ result 1))
     (bits (if (negative? ei)
               (bitwise-not ei)
               ei)
           (bitwise-arithmetic-shift bits -1)))
```

((zero? bits)

result))

Function:**bitwise-first-bit-set** *ei*

[R6RS] Returns the index of the least significant 1 bit in the two's complement representation of *ei*. If *ei* is 0, then - 1 is returned.

Function:**bitwise-bit-set?** *ei1 ei2*

[R6RS] *Ei2* must be non-negative. The *bitwise-bit-set?* procedure returns #t if the *ei2*th bit is 1 in the two's complement representation of *ei1*, and #f otherwise. This is the result of the following computation:

```
(not (zero?
      (bitwise-and
       (bitwise-arithmetic-shift-left 1 ei2)
       ei1)))
```

Function:**bitwise-copy-bit** *ei1 ei2 ei3*

[R6RS] *Ei2* must be non-negative, and *ei3* must be either 0 or 1. The *bitwise-copy-bit* procedure returns the result of replacing the *ei2*th bit of *ei1* by the *ei2*th bit of *ei3*, which is the result of the following computation:

```
(let* ((mask (bitwise-arithmetic-shift-left 1 ei2)))
  (bitwise-if mask
        (bitwise-arithmetic-shift-left ei3 ei2)
        ei1))
```

Function:**bitwise-bit-field** *ei1 ei2 ei3*

[R6RS] *Ei2* and *ei3* must be non-negative, and *ei2* must be less than or equal to *ei3*. The *bitwise-bit-field* procedure returns the number represented by the bits at the positions from *ei2* (inclusive) to *ei3* (exclusive), which is the result of the following computation:

```
(let ((mask
       (bitwise-not
        (bitwise-arithmetic-shift-left -1 ei3))))
  (bitwise-arithmetic-shift-right
   (bitwise-and ei1 mask)
   ei2))
```

Function:**bitwise-copy-bit-field** *ei1 ei2 ei3 ei4*

[R6RS] *Ei2* and *ei3* must be non-negative, and *ei2* must be less than or equal to *ei3*. The *bitwise-copy-bit-field* procedure returns the result of replacing in *ei1* the bits at positions from *ei2* (inclusive) to *ei3* (exclusive) by the corresponding bits in *ei4*, which is the fixnum result of the following computation:

```
(let* ((to    ei1)
       (start ei2)
       (end   ei3)
       (from  ei4)
       (mask1
```

```
  (bitwise-arithmetic-shift-left -1 start))
 (mask2
  (bitwise-not
   (bitwise-arithmetic-shift-left -1 end)))
 (mask (bitwise-and mask1 mask2)))
(bitwise-if mask
      (bitwise-arithmetic-shift-left from
                                start)
      to))
```

Function: **bitwise-arithmetic-shift** *ei1 ei2*

[R6RS] Returns the result of the following computation:

```
(floor (* ei1 (expt 2 ei2)))
```

Function: **bitwise-arithmetic-shift-left** *ei1 ei2*

Function: **bitwise-arithmetic-shift-right** *ei1 ei2*

[R6RS] Ei2 must be non-negative. The *bitwise-arithmetic-shift-left* procedure returns the same result as *bitwise-arithmetic-shift*, and

```
(bitwise-arithmetic-shift-right ei1 ei2)
```

returns the same result as

```
(bitwise-arithmetic-shift ei1 (- ei2)).
```

Function: **bitwise-rotate-bit-field** *ei1 ei2 ei3 ei4*

[R6RS] *Ei2*, *ei3*, *ei4* must be non-negative, *ei2* must be less than or equal to *ei3*, and *ei4* must be non-negative. procedure returns the result of cyclically permuting in *ei1* the bits at positions from *ei2* (inclusive) to *ei3* (exclusive) by *ei4* bits towards the more significant bits, which is the result of the following computation:

```
(let* ((n     ei1)
       (start ei2)
       (end   ei3)
       (count ei4)
       (width (- end start)))
  (if (positive? width)
      (let* ((count (mod count width))
             (field0
              (bitwise-bit-field n start end))
             (field1 (bitwise-arithmetic-shift-left
                      field0 count))
             (field2 (bitwise-arithmetic-shift-right
                      field0
                      (- width count)))
             (field (bitwise-ior field1 field2)))
```

```
  (bitwise-copy-bit-field n start end field))
 n))
```

Function:**bitwise-reverse-bit-field** *ei1 ei2 ei3*

[R6RS] *Ei2* and *ei3* must be non-negative, and *ei2* must be less than or equal to *ei3*. The *bitwise-reverse-bit-field* procedure returns the result obtained from *ei1* by reversing the order of the bits at positions from *ei2* (inclusive) to *ei3* (exclusive).

3.17 (rnrs syntax-case (6))

On Sagittarius Scheme *syntax-case* does not behave as R6RS required. This is why it is **MOSTLY** R6RS implementation. As far as I know the part which is not compromised with R6RS is following expression does not return correct answer:

```
(define-syntax let/scope
 (lambda(x)
  (syntax-case x ()
   ((k scope-name body ...)
    #'(let-syntax
       ((scope-name
         (lambda(x)
          (syntax-case x ()
           ((_ b (... ...))
            #`(begin
               #,@(datum->syntax #'k
                   (syntax->datum #'(b (... ...)))))))))
        body ...)))))
(let ((x 1))
 (let/scope d1
  (let ((x 2))
   (let/scope d2
    (let ((x 3))
     (list (d1 x) (d2 x) x))))))
```

This should return *(1 2 3)* however on Sagittarius it returns *(1 1 3)*. If you want to write a portable program with *syntax-case*, it is better to check in other implementation.

Syntax:**syntax-case** *expression (literal ...) clause*

[R6RS] Each *literal* must be an identifier. Each *clause* must take one of the following two forms.

(pattern output-expression)

(pattern fender output-expression)

Fender and *output-expression* must be expressions.

Pattern is the same as *syntax-rules*. See *(rnrs base (6))* section.

A *syntax-case* expression first evaluates *expression*. It then attempts to match the *pattern* from the first *clause* against the resulting value, which is unwrapped as necessary to perform the match. If the pattern matches the value and no *fender* is present, *output-expression* is evaluated and its value returned as the value of the *syntax-case* expression. If the pattern does not match the value, *syntax-case* tries the second *clause*, then the third, and so on. It is a syntax violation if the value does not match any of the patterns.

If the optional *fender* is present, it serves as an additional constraint on acceptance of a clause. If the *pattern* of a given *clause* matches the input value, the corresponding *fender* is evaluated. If *fender* evaluates to a true value, the clause is accepted; otherwise, the clause is rejected as if the pattern had failed to match the value. Fenders are logically a part of the matching process, i.e., they specify additional matching constraints beyond the basic structure of the input.

Pattern variables contained within a clause's *pattern* are bound to the corresponding pieces of the input value within the clause's *fender* (if present) and *output-expression*. Pattern variables can be referenced only within syntax expressions (see below). Pattern variables occupy the same name space as program variables and keywords.

If the *syntax-case* form is in tail context, the *output-expression*s are also in tail position.

`Syntax:` **syntax** *template*

[R6RS] A *template* is a pattern variable, an identifier that is not a pattern variable, a pattern datum, or one of the following.

(subtemplate ...)

(subtemplate ... . template)

#(subtemplate ...)

A *subtemplate* is a *template* followed by zero or more ellipses.

The value of a *syntax* form is a copy of *template* in which the pattern variables appearing within the template are replaced with the input subforms to which they are bound. Pattern data and identifiers that are not pattern variables or ellipses are copied directly into the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the subtemplate than in the associated subpattern, the input form is replicated as necessary. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis, and for at least one such pattern variable, the subtemplate must be followed by exactly as many ellipses as the subpattern in which the pattern variable appears.

`Function:` **identifier?** *obj*

[R6RS] Returns #t if *obj* is an identifier, i.e., a syntax object representing an identifier, and #f otherwise.

`Function:` **bound-identifier=?** *id1 id2*

[R6RS] *Id1* and *id2* must be identifiers. The procedure *bound-identifier=?* returns #t if given arguments are exactly the same object.

The *bound-identifier=?* procedure can be used for detecting duplicate identifiers in a binding construct or for other preprocessing of a binding construct that requires detecting instances of the bound identifiers.

`Function:` **free-identifier=?** *id1 id2*

[R6RS] *Id1* and *id2* must be identifiers. The *free-identifier=?* procedure returns #t if given arguments are indicating the same bindings.

Function:**syntax->datum** *syntax-object*

[R6RS] Strips all syntactic information from a syntax object and returns the corresponding Scheme datum.

Function:**generate-temporaries** *l*

[R6RS] *L* must be a list or syntax object representing a list-structured form. The number of temporaries generated is the number of elements in *l*. Each temporary is guaranteed to be unique.

NOTE: If you want to create just one temporary symbol and do not think about portability, it's better to use *gensym* in *(sagittarius)* library.

Macro:**with-syntax** *((pattern expression) ...) body*

[R6RS] The *with-syntax* form is used to bind pattern variables, just as let is used to bind variables. This allows a transformer to construct its output in separate pieces, then put the pieces together.

Each *pattern* is identical in form to a syntax-case pattern. The value of each *expression* is computed and destructured according to the corresponding *pattern*, and pattern variables within the *pattern* are bound as with *syntax-case* to the corresponding portions of the value within *body*.

Macro:**quasisyntax** *template*

Auxiliary macro:**unsyntax**

Auxiliary macro: **unsyntax-splicing**

[R6RS] The *quasisyntax* form is similar to syntax, but it allows parts of the quoted text to be evaluated, in a manner similar to the operation of *quasiquote*.

Within a *quasisyntax* template, subforms of *unsyntax* and *unsyntax-splicing* forms are evaluated, and everything else is treated as ordinary template material, as with *syntax*. The value of each *unsyntax* subform is inserted into the output in place of the *unsyntax* form, while the value of each *unsyntax-splicing* subform is spliced into the surrounding list or vector structure. Uses of *unsyntax* and *unsyntax-splicing* are valid only within *quasisyntax* expressions.

A *quasisyntax* expression may be nested, with each *quasisyntax* introducing a new level of syntax quotation and each *unsyntax* or *unsyntax-splicing* taking away a level of quotation. An expression nested within n *quasisyntax* expressions must be within n *unsyntax* or *unsyntax-splicing* expressions to be evaluated.

Function:**syntax-violation** *who message form :optional subform*

[R6RS] *Who* must be #f or a string or a symbol. *Message* must be a string. *Form* must be a syntax object or a datum value. *Subform* must be a syntax object or a datum value. The *syntax-violation* procedure raises an exception, reporting a syntax violation. *Who* should describe the macro transformer that detected the exception. The *message* argument should describe the violation. *Form* should be the erroneous source syntax object or a datum value representing a form. The optional *subform* argument should be a syntax object or datum value representing a form that more precisely locates the violation.

3.18 (rnrs hashtables (6))

The *(rnrs hashtables (6))*library provides a set of operations on hashtables. A hashtable is a data structure that associates keys with values.

Function:**make-eq-hashtable** *:optional k*

Function:**make-eqv-hashtable** *:optional k*

[R6RS] Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys, and compares those keys with *eq? (make-eq-hashtable)* or *eqv?(make-eqv-hashtable)*. If an argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

Function: **make-hashtable** *hash-function equiv :optional k*

[R6RS] *Hash-function* and *equiv* must be procedures. *Hash-function* should accept a key as an argument and should return a non-negative exact integer object. *Equiv* should accept two keys as arguments and return a single value. The *make-hashtable* procedure returns a newly allocated mutable hashtable using *hash-function* as the hash function and *equiv* as the equivalence function used to compare keys. If a third argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

Function: **hashtable?** *obj*

[R6RS] Returns #t if *obj* is a hashtable, #f otherwise.

Function: **hashtable-size** *hashtable*

[R6RS] Returns the number of keys contained in *hashtable* as an exact integer object.

Function: **hashtable-ref** *hashtable key default*

[R6RS] Returns the value in *hashtable* associated with *key*. If *hashtable* does not contain an association for key, *default* is returned.

Function: **hashtable-set!** *hashtable key obj*

[R6RS] Changes *hashtable* to associate *key* with *obj*, adding a new association or replacing any existing association for *key*, and returns unspecified values.

Function: **hashtable-delete!** *hashtable key*

[R6RS] Removes any association for *key* within *hashtable* and returns unspecified values.

Function: **hashtable-contains?** *hashtable key*

[R6RS] Returns #t if *hashtable* contains an association for *key*, #f otherwise.

Function: **hashtable-update!** *hashtable key proc default*

[R6RS] *Proc* should accept one argument, should return a single value. The *hashtable-update!* procedure applies *proc* to the value in *hashtable* associated with *key*, or to default if *hashtable* does not contain an association for *key*. The *hashtable* is then changed to associate *key* with the value returned by *proc*.

Function: **hashtable-copy** *hashtable :optional mutable*

[R6RS] Returns a copy of *hashtable*. If the mutable argument is provided and is true, the returned *hashtable* is mutable; otherwise it is immutable.

Function: **hashtable-clear!** *hashtable :optional k*

[R6RS] Removes all associations from *hashtable* and returns unspecified values.

If a second argument is given, the current capacity of the *hashtable* is reset to approximately *k* elements.

Function: **hashtable-keys** *hashtable*
Function: **hashtable-entries** *hashtable*

[R6RS] Returns a vector of all keys or entries in *hashtable*. The order of the vector is unspecified.

Function: **hashtable-equivalence-function** *hashtable*
Function: **hashtable-hash-function** *hashtable*

[R6RS] Returns the equivalence or hash function used by *hashtable*.

Function: **hashtable-mutable?** *hashtable*

[R6RS] Returns #t if *hashtable* is mutable, otherwise #f.

Function: **equal-hash** *obj*
Function: **string-hash** *obj*
Function: **string-ci-hash** *obj*

Function:**symbol-hash** *obj*

[R6RS] Returns hash value of *obj*. Each procedures return the hash values suitable for *equal?, string=?, string-ci=?* and symbols.

3.19 (rnrs enums (6))

Function:**make-enumeration** *symbol-list*

[R6RS] *Symbol-list* must be a list of symbols. The make-enumeration procedure creates a new enumeration type whose universe consists of those symbols (in canonical order of their first appearance in the list) and returns that universe as an enumeration set whose universe is itself and whose enumeration type is the newly created enumeration type.

Function:**enum-set-universe** *enum-set*

[R6RS] Returns the set of all symbols that comprise the universe of its argument, as an enumeration set.

Function:**enum-set-indexer** *enum-set*

[R6RS] Returns a unary procedure that, given a symbol that is in the universe of *enum-set*, returns its 0-origin index within the canonical ordering of the symbols in the universe; given a value not in the universe, the unary procedure returns #f.

Function:**enum-set-constructor** *enum-set*

[R6RS] Returns a unary procedure that, given a list of symbols that belong to the universe of *enum-set*, returns a subset of that universe that contains exactly the symbols in the list. The values in the list must all belong to the universe.

Function:**enum-set->list** *enum-set*

[R6RS] Returns a list of the symbols that belong to its argument, in the canonical order of the universe of *enum-set*.

Function:**enum-set-member?** *symbol enum-set*
Function:**enum-set-subst?** *enum-set1 enum-set2*
Function:**enum-set=?** *enum-set1 enum-set2*

[R6RS] The *enum-set-member?* procedure returns #t if its first argument is an element of its second argument, #f otherwise.

The *enum-set-subset?* procedure returns #t if the universe of *enum-set1* is a subset of the universe of *enum-set2* (considered as sets of symbols) and every element of *enum-set1* is a member of *enum-set2*. It returns #f otherwise.

The *enum-set=?* procedure returns #t if *enum-set1* is a subset of *enum-set2* and vice versa, as determined by the *enum-set-subset?* procedure. This implies that the universes of the two sets are equal as sets of symbols, but does not imply that they are equal as enumeration types. Otherwise, #f is returned.

Function:**enum-set-union** *enum-set1 enum-set2*
Function:**enum-set-intersection** *enum-set1 enum-set2*
Function:**enum-set-difference** *enum-set1 enum-set2*

[R6RS] *Enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

The *enum-set-union* procedure returns the union of *enum-set1* and *enum-set2*. The *enum-set-intersection* procedure returns the intersection of *enum-set1* and *enum-set2*. The *enum-set-difference* procedure returns the difference of *enum-set1* and *enum-set2*.

Function:**enum-set-complement** *enum-set*

[R6RS] Returns *enum-set*'s complement with respect to its universe.

Function:**enum-set-projection** *enum-set1 enum-set2*

[R6RS] Projects *enum-set1* into the universe of *enum-set2*, dropping any elements of *enum-set1* that do not belong to the universe of *enum-set2*. (If enum-set1 is a subset of the universe of its second, no elements are dropped, and the injection is returned.)

Macro:**define-enumeration** *type-name (symbol ...) constructor-syntax*

[R6RS] The *define-enumeration* form defines an enumeration type and provides two macros for constructing its members and sets of its members.

A *define-enumeration* form is a definition and can appear anywhere any other definition can appear.

Type-name is an identifier that is bound as a syntactic keyword; *symbol ...* are the symbols that comprise the universe of the enumeration (in order).

(*type-name symbol*) checks at macro-expansion time whether the name of *symbol* is in the universe associated with *type-name*. If it is, (*type-name symbol*) is equivalent to *symbol*. It is a syntax violation if it is not.

Constructor-syntax is an identifier that is bound to a macro that, given any finite sequence of the symbols in the universe, possibly with duplicates, expands into an expression that evaluates to the enumeration set of those symbols.

(*constructor-syntax symbol ...*) checks at macro-expansion time whether every *symbol ...* is in the universe associated with *type-name*. It is a syntax violation if one or more is not. Otherwise

(*constructor-syntax symbol ...*)

is equivalent to

((*enum-set-constructor* (*constructor-syntax*))

'(*symbol ...*)).

3.20 (rnrs eval (6))

Function:**eval** *expression environment*

[R6RS] Evaluates *expression* in the specified environment and returns its value. *Expression* must be a syntactically valid Scheme expression represented as a datum value. R6RS requires *envionment* an envitonment which must be created by the *environment* procedure. However on Sagittarius, *environment* can be anything. This behaviour might be fixed in future.

Function:**environment** *import-spec ...*

[R6RS] *Import-spec* must be a datum representing an import spec. The *environment* procedure returns an environment corresponding to *import-spec*.

3.21 (rnrs mutable-pairs (6))

Function:**set-car!** *pair obj*
Function:**set-cdr!** *pair obj*

[R6RS] Store *obj* in the car/cdr field of *pair*. These procedures return unspecified value.

On Sagittarius Scheme, these procedures can modify immutable pairs.

3.22 (rnrs mutable-strings (6))

Function:**string-set!** *string k char*

[R6RS] *K* must be a valid index of *string*. The *string-set!* procedure stores *char* in element

k of *string* and returns unspecified values.

Function:**string-fill!** *string char*

[R6RS] Stores *char* in every element of the given string and returns unspecified values.

Passing an immutable string to these procedures cause an exception with condition type *&assertion* to be raised.

3.23 (rnrs r5rs (6))

Function:**exact->inexact** *z*

Function:**inxact->exact** *z*

[R6RS] These are the same as the *inexact* and *exact* procedures.

Function:**quotient** *n1 n2*

Function:**remainder** *n1 n2*

Function:**modulo** *n1 n2*

[R6RS] Returns the quotient, remainder and modulo of dividing an integer *n1* by an integer *n2*. The result is an exact number only if both *n1* and *n2* are exact numbers.

Macro:**delay** *expression*

[R6RS] The *delay* construct is used together with the procedure *force* to implement lazy evaluation or call by need. (*delay expression*) returns an object called a *promise* which at some point in the future may be asked (by the *force* procedure) to evaluate *expression*, and deliver the resulting value. The effect of *expression* returning multiple values is unspecified.

Macro:**force** *promise*

[R6RS] *Promise* must be a promise. The *force* procedure forces the value of promise. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or “memoized”) so that if it is forced a second time, the previously computed value is returned.

Function:**null-environment** *n*

Function:**scheme-report-environment** *n*

[R6RS] *N* must be the exact integer object 5. These procedures return an environment which is suitable to use with *eval*.

4 Extra Libraries

Sagittarius has its own extension libraries because even R6RS is huge however I know it is not sufficient to write practical program. To support to write it, Sagittarius provides some useful libraries.

4.1 (sagittarius)

This library has Sagittarius specific functions which are not supported in R6RS such as extran file system functions and so.

4.1.1 File system functions

`Function:` **file-size-in-bytes** *filename*
Returns file size of *filename* in bytes. If *filename* does not exist, it raises *&assertion* condition.

`Function:` **file-regular?** *filename*
`Function:` **file-directory?** *filename*
`Function:` **file-symbolic-link?** *filename*
`Function:` **file-readable?** *filename*
`Function:` **file-writable?** *filename*
`Function:` **file-executable?** *filename*
Returns file type or permission of given *filename*.

`Function:` **file-stat-ctime** *filename*
`Function:` **file-stat-mtime** *filename*
`Function:` **file-stat-atime** *filename*
Returns file statistics time in nano sec.The *file-stat-ctime* procedure returns last change time of *filename.* The *file-stat-mtime* returns last modified time of *filename.* The *file-stat-atime* returns last accesse time of *filename.*

`Function:` **create-symbolc-link** *old-filename new-filename*
Creates symbolink of *old-filename* as *new-filename*.

`Function:` **rename-file** *old-filename new-filename*
Renames given *old-filename* to *new-filename*.

`Function:` **create-directory** *path*
`Function:` **delete-directory** *path*
Creates/deletes given directory. If it fails, it raises condition *&assertion*.

`Function:` **read-directory** *path*
Reads directory and returns contents as a string list. If *path* does not exist, it returns #f.

`Function:` **current-directory**
Returns current working directory.

`Function:` **set-current-directory** *path*
Sets current working directory to *path.*

4.1.2 Hashtables

`Function:` **make-equal-hashtable** *:optional k*
`Function:` **make-string-hashtable** *:optional k*
Creates a hashtable. The same as *make-eq-hashtable* and *make-eqv-hashtable*. It uses

equal?/*string=?* as comparing procedure.

Function: **hashtable-keys-list** *hashtable*
Function: **hashtable-values-list** *hashtable*
Returns given *hashtable*'s keys/values. The R6RS required procedures *hashtable-keys* and *hashtable-values* are implemented with these procedures.

Function: **hashtable-type** *hashtable*
Returns *hashtable*'s hash type. The possible return values are *eq, eqv, equal, string* and *general.*

Function: **hashtable-values** *hashtable*
Returns all *hashtable*'s values. This procedure is for consistancy of *hashtable-keys*.

4.1.3 I/O

Function: **port-closed?** *port*
Returns #t if given *port* is closed, otherwise #f.

Function: **write/ss** *obj :optional port*
Writes *obj* to given *port.* If *port* is not given, it uses current output port as output port. This is the same as *write* procedure, but it can handle circular list.

Function: **format** *port string arg ...*
Function: **format** *string arg ...*
[SRFI-28+] Format *arg* accoding to *string. Port* specifies the destination; if it is an output port, the formatted result is written to it; if it is #t, the result is written to current output port; if it is #f, the formatted result is returned as a string. *Port* can be omitted and the result is the same as when #f is given.

String is a string that contains format directives. A format directive is a character sequence begins with tilda '~', and ends with some specific characters. A format directive takes the corresponding arg and formats it. The rest of string is copied to the output as is.

(*format #f "the answer is ~a" 48) => "the anser is 48"*

The format directive can take one or more parameters, separated by comma characters. A parameter may be an integer or a character; if it is a character, it should be preceded by a quote character. Parameter can be omitted, in such case the system default value is used. The interpretation of the parameters depends on the format directive.

Furthermore, a format directive can take two additional flags: atmark '@' and colon ':'. One or both of them may modify the behavior of the format directive. Those flags must be placed immediately before the directive character.

The following complete list of the supported directives. Either upper case or lower case character can be used for the format directive; usually they have no distinction, except noted.

- *~mincol,colinc,minpad,padchar,maxcol***A**

  Ascii output. The corresponding argument is printed by display. If an integer *mincol* is given, it specifies the minimum number of characters to be output; if the formatted result is shorter than mincol, a whitespace is padded to the right (i.e. the result is left justified).

  The *colinc*, *minpad* and *padchar* parameters control, if given, further padding. A character padchar replaces the padding character for the whitespace. If an integer minpad is given and greater than 0, at least minpad padding character is used, regardless of the resulting width. If an integer *colinc* is given, the padding character is added (after *minpad*) in chunk of *colinc* characters, until the entire width exceeds *mincol*.

  If atmark-flag is given, the format result is right justified, i.e. padding is added to the left.

The *maxcol* parameter, if given, limits the maximum number of characters to be written. If the length of formatted string exceeds maxcol, only *maxcol* characters are written. If colon-flag is given as well and the length of formatted string exceeds *maxcol*, *maxcol* - 4 characters are written and a string " ..." is attached after it.

(format #f "|~a|" "oops") → *"|oops|"*

(format #f "|~10a|" "oops") → *"|oops |"*

(format #f "|~10@a|" "oops") → *"| oops|"*

(format #f "|~10,,,'@a|" "oops")* → *"|******oops|"*

(format #f "|~,,,,10a|" '(abc def ghi jkl)) → *"|(abc def gh|"*

(format #f "|~,,,,10:a|" '(abc def ghi jkl)) → *"|(abc de ...|"*

- *~mincol,colinc,minpad,padchar,maxcol***S**

  S-expression output. The corresponding argument is printed by write. The semantics of parameters and flags are the same as ~**A** directive.

  (format #f "|~s|" "oops") ⇒ *"|\"oops\"|"*

  (format #f "|~10s|" "oops") ⇒ *"|\"oops\" |"*

  (format #f "|~10@s|" "oops") ⇒ *"| \"oops\"|"*

  (format #f "|~10,,,'@s|" "oops")* ⇒ *"|****\"oops\"|"*

- *~mincol,padchar,commachar,interval***D**

  Decimal output. The argument is formatted as an decimal integer. If the argument is not an integer, all parameters are ignored and it is formatted by ~**A** directive.

  If an integer parameter *mincol* is given, it specifies minimum width of the formatted result; if the result is shorter than it, padchar is padded on the left (i.e. the result is right justified). The default of *padchar* is a whitespace.

  (format #f "|~d|" 12345) ⇒ *"|12345|"*

  (format #f "|~10d|" 12345) ⇒ *"| 12345|"*

  (format #f "|~10,'0d|" 12345) ⇒ *"|0000012345|"*

  If atmark-flag is given, the sign '+' is printed for the positive argument.

  If colon-flag is given, every *interval*-th digit of the result is grouped and commachar is inserted between them. The default of *commachar* is ',', and the default of *interval* is 3.

  (format #f "|~:d|" 12345) ⇒ *"|12,345|"*

  (format #f "|~,,'_,4:d|" -12345678) ⇒ *"|-1234_5678|"*

- *~mincol,padchar,commachar,interval***B**

  Binary output. The argument is formatted as a binary integer. The semantics of parameters and flags are the same as the ~**D** directive.

- *~mincol,padchar,commachar,interval***D**

  Octet output. The argument is formatted as a octal integer. The semantics of parameters and flags are the same as the ~**D** directive.

- *~mincol,padchar,commachar,interval***X**

  *~mincol,padchar,commachar,interval***x**

Hexadecimal output. The argument is formatted as a hexadecimal integer. If ‘X’ is used, upper case alphabets are used for the digits larger than 10. If ‘x’ is used, lower case alphabets are used. The semantics of parameters and flags are the same as the ~**D** directive.

(format #f "~8,'0x" 259847592) ⇒ "0f7cf5a8"

(format #f "~8,'0X" 259847592) ⇒ "0F7CF5A8"

Note: The format procedures's implementation and most of documentation is quoted from Gauche.

Function: **make-codec** *symbol getc putc data*

Creates a custom codec. *Symbol* is the name of the codec.*Getc* and *putc* must be procedures.*Data* is an user data used in *getc* and *putc*.

Getc must take 4 arguments, *input-port, error-handling-mode, check-bom?* and *userdata*. *Input-port* is binary input port. *Error-handling-mode* is symbol can be *ignore, raise* or *replace* depending on a transcoder which uses this custom codec. *Check-bom?* is boolean, if *getc* is being called first time, it is #t, otherwise #f. *Userdata* is user defined data which is given when the codec is created.

The basic process of *getc* is reading binary data from *input-port* and converting the data to UCS4. Returning UCS4 must be integer and does not have to be 4 byte.

Putc must take 4 arguments, *output-port, char, error-handling-mode* and *userdata*. *Output-port* is binary output port. *Char* is character object which needs to be converted from UCS4. *Error-handling-mode* is symbol can be *ignore, raise* or *replace* depending on a transcoder which uses this custom codec.*Userdata* is user defined data which is given when the codec is created.

The basic process of *putc* is converting given UCS4 charactner to target encoding data and putting it to *output-port* as binary data.

For sample implementation, see sitelib/encoding directory. You can find some custom codecs.

4.1.4 Keywords

Sagittarius has keyword objects which starts with '**:**'. It has almost the same feature as symbol, however it can not be bounded with any values. It can be used when variable is bounded by *define-with-key* (see (sagittarius control) library).

Function: **make-keyword** *symbol*

Creates a new keyword from *symbol*.

Function: **keyword?** *obj*

Returns #t if *obj* is keyword, otherwise #f.

4.1.5 Environment variables

These procedures are used to implement SRFI-98. For portablity it is better to use SRFI-98.

Function: **getenv** *string*

Returns environment variable named *string*. If nothing is found, returns #f.

Function: **setenv** *string1 string2*

Sets environment variable *string1* with value *string2*.

Function: **getenv-alist**

Returns all environemt variables as an associative list.

4.1.6 Bytevector operations

Function: **bytevector->integer** *bv :optional start end*

Converts given bytevector *bv* to exact integer. If optional argument *start* is given, convertion starts with index *starts*. If optional argument *end* is given, convertion ends by index *end.*

Function: **integer->bytevector** *ei*

Ei must be exact integer. Converts exact integer *ei* to bytevector.

4.1.7 List operations

Function: **circular-list?** *list*

Function: **dotted-list** *list*

[SRFI-1] Returns #t if *list* is circular/dotted list, otherwise #f.

Function: **acons** *obj1 obj2 obj3*

[SRFI-1] Returns *(cons (cons obj1 obj2) obj3)*. Useful to put an entry at the head of an associative list.

Function: **append!** *list ...*

[SRFI-1] Returns a list consisting of the elements of the first *list* followed by the elements of the other lists. The cells in the lists except the last one may be reused to construct the result. The last argument may be any object.

Function: **reverse!** *list ...*

[SRFI-1] Returns a list consisting of the elements of *list* in reverse order. The cells of list may be reused to construct the returned list.

4.1.8 Vector operations

Function: **vector-copy** *vector :optional start end fill*

Copies a vector vector. Optional *start* and *end* arguments can be used to limit the range of *vector* to be copied. If the range specified by *start* and *end* falls outside of the original vector, the *fill* value is used to fill the result vector.

4.2 (sagittarius regex)

As most of script language have own regular expression mechanism, Sagittarius also has own regular expression library. It is influenced by Java's regular expression, so there are a lot of differences between the most famous Perl regular expression(perlre). This feature may be changed, if R7RS large requires Perl like regular expression.

Following examples show how to use Sagittarius's regular expression.

```
;; For Perl like
(cond ((looking-at (regex "^hello\\s*(.+)") "hello world!")
       => (lambda (m) (m 1))) ;; return "world!"
;; For Java like
(cond ((matches (regex "(\\w+?)\\s*(.+)") "123hello world!")) ;; this won't match
      (else "incovenient eh?"))
```

The *matches* procedure is total match, so it ignores boundary matcher '^' and '$'. The *looking-at* procedure is partial match, so it works as if perlre. Sagittarius support most of Java regular expression constructs, however POSIX character classes and Classes for Unicode blocks and

categories are not supported yet.

`Function:` **regex** *string :optional flags*

String must be regular expression. Returns compiled regular expression. *Flags*' descriptions are the end of this section. The following table is the supported regular expression constructs.

Construct	Matches
	Characters
x	The character x
\\	The backslash character
\0n	The character with octal value 0n (0 <= n <= 7)
\0nn	The character with octal value 0nn (0 <= n <= 7)
\0mnn	The character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7)
\xhh	The character with hexadecimal value 0xhh
\uhhhh	The character with hexadecimal value 0xhhhh
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage-return character ('\u000D')
\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\cx	The control character corresponding to x
	Character classes
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
	Predefined character classes
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]
	Boundary matchers
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input
	Greedy quantifiers
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
X{n,}	X, at least n times

X{n,m}	X, at least n but not more than m times
	Reluctant quantifiers
X??	X, once or not at all
X*?	X, zero or more times
X+?	X, one or more times
X{n}?	X, exactly n times
X{n,}?	X, at least n times
X{n,m}?	X, at least n but not more than m times
	Possessive quantifiers
X?+	X, once or not at all
X*+	X, zero or more times
X++	X, one or more times
X{n}+	X, exactly n times
X{n,}+	X, at least n times
X{n,m}+	X, at least n but not more than m times
	Logical operators
XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group
	Back references
\n	Whatever the nth capturing group matched
	Quotation
\	Nothing, but quotes the following character
\Q	Nothing, but quotes all characters until \E
\E	Nothing, but ends quoting started by \Q
	Special constructs (non-capturing)
(?:X)	X, as a non-capturing group
(?idmsux-idmsux)	Nothing, but turns match flags on - off
(?idmsux-idmsux:X)	X, as a non-capturing group with the given flags on - off
(?=X)	X, via zero-width positive lookahead
(?!X)	X, via zero-width negative lookahead
(?<=X)	X, via zero-width positive lookbehind
(?<!X)	X, via zero-width negative lookbehind
(?>X)	X, as an independent, non-capturing group

`Function:` **matches** *regex string*
`Function:` **looking-at** *regex string*

Regex must be regular expression object. Returns closure if *regex* matches input *string*. The *matches* procedure attempts to match the entire input string against the pattern of *regex*. The *looking-at* procedure attempts to match the input string against the pattern of *regex*.

`Function:` **regex-replace-first** *regex string1 string2*
`Function:` **regex-replace-all** *regex string1 string2*

Regex must be regular expression object. Replaces part of *string1* where *regex* matches to *string2*. The *regex-replace-first* procedure replaces the first match. The *regex-replace-all* procedure replaces all strings which matches *regex*.

4.2.1 Low level API for regular expression

The above procedures are wrapped User level API. However, you might want to use low level API directory when you re-use matcher and recursively find pattern from input. For that purpose, you need to use low level API directory.

NOTE: This API might be changed in future depending on R7RS large.

`Function:` **regex-pattern?** *obj*
Returns #f if *obj* is regular expression object, otherwise #f.

`Function:` **regex-matcher?** *obj*
Returns #f if *obj* is matcher object, otherwise #f.

`Function:` **compile-regex** *string :optional flags*
The same as *regex* procedure.

`Function:` **regex-matcher** *regex string*
Regex must be regular expression object. Returns matcher object.

`Function:` **regex-matches** *matcher*
Matcher must be matcher object. Returns #t if *matcher* matches the entire input string against input pattern, otherwise #f.

`Function:` **regex-looking-at** *matcher*
Matcher must be matcher object. Returns #t if *matcher* matches the input string against input pattern, otherwise #f.

`Function:` **regex-find** *matcher :optional start*
Matcher must be matcher object. Resets *matcher* and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index if optional argument is given otherwise from the beginning.

4.2.2 Regular expression flags

Regular expression compiler can take following flags.

- CASE-INSENSITIVE

  Enables case-insensitive matching.

- COMMENTS

  Permits whitespace and comments in pattern.

- MULTILINE

  Enables multiline mode.

- LITERAL

  Enables literal parsing of the pattern

- DOTAIL

  Enables dotall mode.

- UNICODE-CASE

  Enables Unicode-aware case folding.

4.3 (sagittarius socket)

This section describes low level socket API on Sagittarius. The APIs are mostly the same signature as Ypsilon and mosh. The following example is simple echo server, it receives input from a client and just returns it to the client.

The example program is from example/socket/echo.scm.

```
(import (rnrs) (sagittarius socket))
;; creates echo server socket with port number 5000
(define echo-server-socket (make-server-socket "5000"))
;; addr is client socket
(let loop ((addr (socket-accept echo-server-socket)))
  (call-with-socket addr
   (lambda (sock)
     ;; socket-port creates binary input/output port
     ;; make it transcoded port for convenience.
     (let ((p (transcoded-port (socket-port sock)
                               ;; on Sagittarius Scheme native-transcoder
                               ;; uses utf8 codec for ASCII compatibility.
                               ;; For socket programming it might be better
                               ;; to specify eol-style with crlf.
                               ;; But this sample just shows how it goes.
                               (native-transcoder))))
       (call-with-port p
         (lambda (p)
           (put-string p "please type something\n\r")
           (put-string p "> ")
           ;; gets line from client.
           (let lp2 ((r (get-line p)))
             (unless (eof-object? r)
               (print "received: " r)
               ;; just returns message from client.
               ;; NB: If client type nothing, it'll throw assertion-violation.
               (put-string p r)
               (put-string p "\r\n> ")
               ;; waits for next input.
               (lp2 (get-line p)))))))))
  ;; echo server waits next connection.
  (loop (socket-accept echo-server-socket)))
```

Function: **make-client-socket** *node service :optional ai_family ai_socktype ai_flags ai_protocol*

Node and *service* must be string or #f. *Node* Other optional arguments must be exact integer. Returns a client socket connected to an Internet address. The Internet address is identified by *node* and *service*. The *make-client-socket* uses getaddrinfo(3) to look it up. The arguments *node, service, ai-family, ai-socktype, ai-flags* and *ai-protocol* are passed to getaddrinfo(3) as corresponding

parameters. For more detail, see reference of getaddrinfo(3).

Node is a network address, ex) “www.w3.org”, “localhost”, “192.168.1.1”.

Service is a network service, ex) “http”, “ssh”, “80” “22”.

Ai-family is an address family specifier. Default value is AF_INET. Predefined specifiers are listed below.

- AF_INET
- AF_INET6
- AF_UNSPEC

Ai-sockettype is a socket type specifier. Default value is SOCK_STREAM. Predefined specifiers are listed below.

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW

Ai-flags is an additional options specifier. Default value is AI_V4MAPPED + AI_ADDRCONFIG. Predefined specifiers are listed below.

- AI_ADDRCONFIG
- AI_ALL
- AI_CANONNAME
- AI_NUMERICHOST
- AI_NUMERICSERV
- AI_PASSIVE
- AI_V4MAPPED

Ai-protocol is a protocol specifier. Default value is 0. Predefined specifiers are listed below.

- IPPROTO_TCP
- IPPROTO_UDP
- IPPROTO_RAW

`Function:` **make-server-socket** *service :optional ai_family ai_socktype ai_protocol*

Service must be string or #f. Other optional arguments must be exact integer. Returns a server socket waiting for connections. The argument details are the same as the *make-client-socket.*

`Function:` **socket?** *obj*

Returns #t if *obj* is socket object, otherwise #f.

`Function:` **socket-port** *socket*

Socket must be a socket object. Returns a binary input/output port associated with *socket*.

`Function:` **call-with-socket** *socket proc*

Socket must be a socket object. *Proc* must accept one argument. The *call-with-socket* calls a procedure with *socket* as an argument. This procedure is analogy with *call-with-port*.

`Function:` **shutdown-output-port** *port*

Port must be associated with a socket. The *shutdown-output-port* shutdowns output connection of a socket associated with *port*.

Function:**socket-accept** *socket*
Function:**socket-recv** *socket ei ei*
Function:**socket-send** *socket bytevector ei*

Socket must be a socket object. These procedures are thin wrapper of accept(2), recv(2) and send(2). For more detail, see these manuals.

Function:**socket-shutdown** *socket how*

Socket must be a socket object. *How* must be one of the SHUT_RD, SHUT_WR or SHUT_RDWR. The *socket-shutdown* shutdowns *socket.*

SHUT_RD shutdowns input.

SHUT_WR shutdowns output.

SHUT_RDWR shutdowns input and output.

Function:**socket-close** *socket*

Socket must be a socket object. Closes *socket*.

4.3.1 Platform dependent values

Following platform dependent values are defined in Sagittarius. Each value is defined as an exact integer or if the platform does not support the value it is defined as #f. And these are corresponding C header file definition.

- AF_UNSPEC
- AF_INET
- AF_INET6
- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW
- SOCK_RDM
- SOCK_SEQPACKET
- AI_PASSIVE
- AI_CANONNAME
- AI_NUMERICHOST
- AI_V4MAPPED
- AI_ALL
- AI_ADDRCONFIG
- SHUT_RD
- SHUT_WR
- SHUT_RDWR
- MSG_OOB
- MSG_PEEK
- MSG_DONTROUTE

- MSG_CTRUNC
- MSG_PROBE
- MSG_TRUNC
- MSG_DONTWAIT
- MSG_EOR
- MSG_WAITALL
- MSG_FIN
- MSG_SYN
- MSG_CONFIRM
- MSG_RST
- MSG_ERRQUEUE
- MSG_NOSIGNAL
- MSG_MORE
- MSG_EOF

4.4 (sagittarius control)

This library provides some useful macros using Sagittarius specific functions.

Macro:**define-optional** (*name arg ... (optional opt-arg ...)) body ...*

The *define-optional* form is a definition used to create variable bindings and may appear anywhere other definitions may appear. This can take explicit optional arguments. For example:

(define-optional (buzz arg1 arg2 (optional (opt2 'option) opt2)) <body>)

It seems similar with usual *define* form, after the *optional* keyword arguments can be optional arguments and it can take default value; on the above example *opt1* has *'option* as default value. If an optional argument does not have default value #f is bound.

Macro:**let-optionals*** *restargs (var-spec ...) body ...*
Macro:**let-optionals*** *restargs (var-spec ... . restvar) body ...*

Given a list of values *restargs*, binds variables according to *var-spec*, then evaluates body.

Var-spec can be either a symbol, or a list of two elements and its car is a symbol. The symbol is the bound variable name. The values in *restargs* are bound to the symbol in order. If there are not as many values in *restargs* as var-spec, the rest of symbols are bound to the default values, determined as follows: If var-spec is just a symbol, the default value is undefined. If *var-spec* is a list, the default value is the result of evaluation of the second element of the list. In the latter case the second element is only evaluated when there are not enough arguments. The binding proceeds in the order of *var-spec*, so the second element may refer to the bindings of previous *var-spec*.

In the second form, *restvar* must be a symbol and bound to the list of values whatever left from *restargs* after binding to *var-spec*.

It is not an error if *restarg* has more values than *var-specs*. The extra values are simply ignored in the first form.

Macro:**get-optional** *restargs default*
Macro:**get-optional** *restargs default test*

This is a short version of *let-optionals** where you have only one optional argument. Given the optional argument list *restargs*, this macro returns the value of optional argument if one is given, or the result of default otherwise. If latter form is used, *test* must be procedure which takes one argument and it will be called to check the given argument. Default is not evaluated unless *restargs* is an empty list.

Macro: **let-keywords** *restargs (var-spec ...) body ...*
Macro: **let-keywords** *restargs (var-spec ... . restvar) body ...*

This macro is for keyword arguments. *Var-spec* can be one of the following forms:

(*symbol expr*)

If the *restrag* contains keyword which has the same name as *symbol*, binds symbol to the corresponding value. If such a keyword doesn't appear in *restarg*, binds symbol to the result of *expr.*

(*symbol keyword expr*)

If the *restarg* contains keyword *keyword*, binds symbol to the corresponding value. If such a keyword doesn't appear in *restarg*, binds symbol to the result of *expr.*

The default value *expr* is only evaluated when the keyword is not given to the *restarg*.

If you use the first form, *let-keyword* throws an error when *restarg* contains a keyword argument that is not listed in *var-specs*. When you want to allow keyword arguments other than listed in *var-specs*, use the second form.

In the second form, *restvar* must be either a symbol or #f. If it is a symbol, it is bound to a list of keyword arguments that are not processed by var-specs. If it is #f, such keyword arguments are just ignored.

```
(define (proc x . options)
  (let-keywords options ((a 'a)
                         (b :beta 'b)
                         (c 'c)
                         . rest)
    (list x a b c rest)))
```

(proc 0) → (0 a b c ())

(proc 0 :a 1) → (0 1 b c ())

(proc 0 :beta 1) → (0 a 1 c ())

(proc 0 :beta 1 :c 3 :unknown 4) → (0 a 1 3 (:unknown 4))

Macro: **let-keywords*** *restargs (var-spec ...) body ...*
Macro: **let-keywords*** *restargs (var-spec ... . restvar) body ...*

Like *let-keywords*, but the binding is done in the order of *var-specs*. So each *expr* can refer to the variables bound by preceding *var-specs*.

These macro's are originally from Gauche.

Macro: **define-with-key** *(name arg ... :key (ver-spec ...)) body ...*
Macro: **define-with-key** *(name arg ... :key (ver-spec ... . restvar)) body ...*
Macro: **define-with-key** *(name arg ... :key (ver-spec ...) :allow-other-keys opt ...) body ...*
Macro: **define-with-key** *(name arg ... :key (ver-spec .... restvar) :allow-other-keys opt ...) body ...*

The *define-with-key* is similar with the form *define*, it uses *let-keywors** inside. If the third or fourth form is used, it can receive non-defined keywords. The macro will be expanded approximately like this:

(define-with-key (name arg0 arg1 :key key1 (key2 'value) :allow-other-keys opt) body …)

=> (define (name arg0 arg1 . optional)

(let-keywords* optional (key1

(key2 #f)

. opt)

body …))

`Macro:` **begin0** *exp0 exp1 ...*

Evaluate *exp0*, *exp1*, ..., then returns the result(s) of *exp0*.

`Macro:` **let1** *var expr body ...*

A convenient macro when you have only one variable. Expanded as follows:

(let ((*var expr*)) *body ...*)

5 Supporting SRFIs

SRFI is a great libraries, so there is no reason not to support. Without exception Sagittarius also supports several SRFIs. The following list is the supported SRFI. Documents are not written for now. So if you need to refer the functions, please look for SRFI's site. I might write it later.

SRFI number	Library name
SRFI-0	(srfi :0 cond-expand)
SRFI-1	(srfi :1 lists)
SRFI-2	(srfi :2 and-let*)
SRFI-6	(srfi :6 basic-string-ports)
SRFI-8	(srfi :8 receive)
SRFI-13	(srfi :13 strings)
SRFI-14	(srfi :14 char-set)
SRFI-23	(srfi :23 error)
SRFI-26	(srfi :26 cut)
SRFI-39	(srfi :39 parameters)
SRFI-41	(srfi :41 streams)
SRFI-42	(srfi :42 eager-comprehensions)
SRFI-64	(srfi :64 testing)
SRFI-98	(srfi :98 os-environment-variables)

Each library can be imported like this:

```
(import (srfi :1))
```

So you don't have to type the long name.