

STklos Reference Manual

version 0.57

Erick Gallesio

Introduction

This document provides a complete list of procedures and special forms implemented in version 0.57 of STKLOS. Since STKLOS is (nearly) compliant with the language described in the Revised⁵ Report on the Algorithmic Language Scheme [R5RS], the organization of this manual follows closely the one of *R⁵RS*.

1 Overview of STklos

STKLOS is the successor of STK[Gallesio95], a Scheme interpreter which was tightly connected to the Tk graphical toolkit [Ousterhout91]. STK had an object layer which was called STKLOS. At the time, there were two different names for the base Scheme interpreter and its object layer, because the latter was not mandatory. For instance, when programming a GUI application you could access the widgets at the (low) Tk level, or access them using a neat hierarchy of classes.

Since, the object layer of is now more closely integrated with the language now, the new system has been renamed in STKLOS and STK is used to designate the old system.

Compatibility with STK: STKLOS has been completely rewritten and as a consequence, due to new implementation choices, old STK programs are not fully compatible with the new system. However, these changes are very minor and adapting a STK program for the STKLOS system is generally quite easy. The only programs which needs heavier work are programs which used Tk without objects, since the new preferred GUI system is now based on GTK+ [GTK01]. Programmers used to GUI programming using STKLOS classes will find that both system, even if not identical in every points, share the same philosophy.

2 Lexical Conventions

2.1 Identifiers

Identifiers in STKLOS with a colon “:” as first (or last) character are considered as keywords, see [keywords], page 49.

2.2 Comments

There are three types of comments in STKLOS:

- a semicolon “;” indicates the start of a comment. This kind of comment extends to the end of the line (as described in *R⁵RS*).
- multi-lines comment use the classical Lisp convention: a comment begins with “#’” and ends with ‘#’.
- comments can also be introduced by “#!”. Such a comment extends to the end of the line which introduces it. This extension is particularly useful for building STKLOS scripts. On most Unix implementations, if the first line of a script looks like this:

```
#!/usr/local/bin/stklos -file
```

then the script can be started directly as if it was a binary program. STKLOS is loaded behind the scene and reads and executes the script as a Scheme program. Note that, as a special case, the sequences `#!key`, `#!optional` and `#!rest` are respectively converted to the STKLOS keywords `:key`, `:optional`. This permits to Scheme lambdas, which accepts keywords and optional arguments, to be compatible with DSSSL lambdas [DSSSL-96].

2.3 Other Notations

STK accepts all the notations defined in *R⁵RS* plus

- `[]` Brackets are equivalent to parentheses. They are used for grouping and to build lists. A list opened with a left square bracket must be closed with a right square bracket ([\[lists\]](#), page 25).
- `:` a colon at the beginning or the end of a symbol introduces a keyword. Keywords are described in section [\[keywords\]](#), page 49.
- `#n=` is used to represent circular structures. The value given of *n* must be a number. It is used as a label, which can be referenced later by a `#n#` notation (see below). The scope of the label is the expression being read by the outermost `read`.
- `#n#` is used to reference some object previously labeled by a `#n=` notation; that is, `#n#` represents a pointer to the object labeled exactly by `#n=`. For instance, the object returned by the following expression

```
(let* ((a (list 1 2))
      (b (append '(x y) a)))
  (list a b))
```

can also be represented in this way:

```
(#0=(1 2) (x y . #0#))
```

- `{ } TODO//: DESCRIBE HERE BRACE NOTATION`

3 Basic Concepts

See the original *R⁵RS* document for more informations.

TODO//: DESCRIBE THE VOID VALUE

4 Expressions

This chapter describes the main syntaxes available in STKLOS. For constructions, which are similar to *R⁵RS*, the descriptions are given very succinctly here for reference. See the *R⁵RS* for a complete description.

4.1 Literal expressions

`quote <datum>`

R⁵RS Syntax

`'<datum>`

R⁵RS Syntax

The quoting mechanism is identical to *R⁵RS*, except that keywords constants evaluate "to themselves" as numerical constants, string constants, character constants, and boolean constants

```
'"abc"      ⇒ "abc"
"abc"       ⇒ "abc"
'145932     ⇒ 145932
145932     ⇒ 145932
'#t        ⇒ #t
#t         ⇒ #t
:foo       ⇒ :foo
':foo     ⇒ :foo
```

Note: *R⁵RS* requires to quote constant lists and constant vectors. This is not necessary with STKLOS.

4.2 Procedures

lambda *<formals>* *<body>*

STKLOS Syntax

A lambda expression evaluates to a procedure. STklos lambda expression have been extended to allow an optional and keyword parameters. *<formals>* should have one of the following forms:

- (*<variable1>* ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables. This form is identical to R^5RS .
- *<variable>*: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the *<variable>*. This form is identical to R^5RS .
- (*<variable1>* ... *<variablen>* . *<variablen+1>*): If a space-delimited period precedes the last variable, then the procedure takes *n* or more arguments, where *n* is the number of formal arguments before the period (there must be at least one). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments. This form is identical to R^5RS .
- (*<variable1* ... *<variablen>* [:*optional* ...] [:*rest* ...] [:*key* ...]) This form is specific to STKLOS and allows to have procedure with optional and keyword parameters. The form *:optional* allows to specify optional parameters. All the parameters specified after *:optional* to the end of *<formals>* (or until a *:rest* or *:key*) are optional parameters. An optional parameter can be declared as:
 - *variable*: if a value is passed when the procedure is called, it will be stored in the binding of the corresponding variable, otherwise the value **#f** will be stored in it.
 - (*variable value*): if a value is passed when the procedure is called, it will be stored in the binding of the corresponding variable, otherwise *value* will be stored in it.
 - (*variable value test?*): if a value is passed when the procedure is called, it will be stored in the binding of the corresponding variable, otherwise *value* will be stored in it. Furthermore, *test?* will be given the value **#t** if a value is passed for the given variable, otherwise *test?* is set to **#f**

Hereafter are some examples using *:optional* parameters

```
((lambda (a b :optional c d) (list a b c d)) 1 2)
  ⇒ (1 2 #f #f)
((lambda (a b :optional c d) (list a b c d)) 1 2 3)
  ⇒ (1 2 3 #f)
((lambda (a b :optional c (d 100)) (list a b c d)) 1 2 3)
  ⇒ (1 2 3 100)
((lambda (a b :optional c (d #f d?)) (list a b c d d?)) 1 2 3)
  ⇒ (1 2 3 #f #f)
```

The form *:rest* parameter is similar to the dot notation seen before. It is used before an identifier to collect the parameters in a single binding:

```
((lambda (a :rest b) (list a b)) 1)
  ⇒ (1 ())
((lambda (a :rest b) (list a b)) 1 2)
  ⇒ (1 (2))
((lambda (a :rest b) (list a b)) 1 2 3)
  ⇒ (1 (2 3))
```

The form `:key` allows to use keyword parameter passing. All the parameters specified after `:key` to the end of `<formals>` are keyword parameters. A keyword parameter can be declared using the three forms given for optional parameters. Here are some examples illustrating how to declare and how to use keyword parameters:

```
((lambda (a :key b c) (list a b c)) 1 :c 2 :b 3)
    ⇒ (1 3 2)
((lambda (a :key b c) (list a b c)) 1 :c 2)
    ⇒ (1 #f 2)
((lambda (a :key (b 100 b?) c) (list a b c b?)) 1 :c 2)
    ⇒ (1 100 2 #f)
```

At last, here is an example showing `:optional`, `:rest` and `:key` parameters

```
(define f (lambda (a :optional b :rest c :key d e)
            (list a b c d e)))
(f 1) ⇒ (1 #f () #f #f)
(f 1 2) ⇒ (1 2 () #f #f)
(f 1 2 :d 3 :e 4) ⇒ (1 2 (:d 3 :e 4) 3 4)
(f 1 :d 3 :e 4) ⇒ (1 #f (:d 3 :e 4) 3 4)
```

closure? *obj*

STKLOS Procedure

Returns `#t` if *obj* is a procedure created with the `lambda` syntax and `#f` otherwise.

case-lambda *<clause> ...*

STKLOS Syntax

Each *<clause>* should have the form `(<formals> <body>)`, where *<formals>* is a formal arguments list as for `lambda`. Each *<body>* is a *<tail-body>*, as defined in *R⁵RS*.

A `case-lambda` expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as procedures resulting from `lambda` expressions. When the procedure is called with some arguments *v₁ ... v_k*, then the first *<clause>* for which the arguments agree with *<formals>* is selected, where agreement is specified as for the *<formals>* of a `lambda` expression. The variables of *<formals>* are bound to fresh locations, the values *v₁ ... v_k* are stored in those locations, the *<body>* is evaluated in the extended environment, and the results of *<body>* are returned as the results of the procedure call.

It is an error for the arguments not to agree with the *<formals>* of any *<clause>*.

This form is defined in SRFI-16 ("Syntax for procedures of variable arity")

```
(define plus
  (case-lambda
    ((()) 0)
    ((x) x)
    ((x y) (+ x y))
    ((x y z) (+ (+ x y) z))
    (args (apply + args))))

(plus) ⇒ 0
(plus 1) ⇒ 1
(plus 1 2 3) ⇒ 6

((case-lambda
  ((a) a)
  ((a b) (* a b)))
 1 2 3) ⇒ error
```

4.3 Conditionals

if <test> <consequent> <alternate>

R⁵RS Syntax

if <test> <consequent>

R⁵RS Syntax

An if expression is evaluated as follows: first, <test> is evaluated. If it yields a true value, then <consequent> is evaluated and its value(s) is(are) returned. Otherwise <alternate> is evaluated and its value(s) is(are) returned. If <test> yields a false value and no <alternate> is specified, then the result of the expression is *void*.

```
(if (> 3 2) 'yes 'no)      ⇒ yes
(if (> 2 3) 'yes 'no)      ⇒ no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                ⇒ 1
```

cond <clause1> <clause2> ...

R⁵RS Syntax

In a cond, each <clause> should be of the form

```
(<test> <expression1> ...)
```

where <test> is any expression. Alternatively, a <clause> may be of the form

```
(<test> ⇒ <expression>)
```

The last <clause> may be an "else clause," which has the form

```
(else <expression1> <expression2> ...)
```

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 result(s) of the last <expression> in the <clause> is(are) returned as the result(s) 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 that accepts one argument; this procedure is then called on the value of the <test> and the value(s) returned by this procedure is(are) returned by the cond expression. If all <test>s evaluate to false values, and there is no else clause, then the result of the conditional expression is *void*; if there is an else clause, then its <expression>|s are evaluated, and the value(s) of the last one is(are) returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))          ⇒ greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))          ⇒ equal

(cond ((assv 'b '((a 1) (b 2))) ⇒ cadr)
      (else #f))              ⇒ 2
```

case <key> <clause1> <clause2> ...

R⁵RS Syntax

In a case, each <clause> should have the form

```
((<datum1> ...) <expression1> <expression2> ...),
```

where each <datum> is an external representation of some object. All the <datum>s must be distinct. The last <clause> may be an "else clause," which has the form

```
(else <expression1> <expression2> ...).
```

A case expression is evaluated as follows. <key> is evaluated and its result is compared against each <datum>. If the result of evaluating <key> is equivalent (in the sense of `eqv?`) to a <datum>, then the expressions in the corresponding <clause> are evaluated from left to right and the result(s) of the last expression in the <clause> is(are) returned as the result(s) of the case expression. If the result of evaluating <key> is different from every <datum>, then if there is an else clause its expressions are evaluated and the result(s) of the last is(are) the result(s) of the case expression; otherwise the result of the case expression is *void*.

```
(case (* 2 3)
      ((2 3 5 7) 'prime)
      ((1 4 6 8 9) 'composite))    ⇒ composite
(case (car '(c d))
      ((a) 'a)
      ((b) 'b))                    ⇒ void
(case (car '(c d))
      ((a e i o u) 'vowel)
      ((w y) 'semivowel)
      (else 'consonant))           ⇒ consonant
```

and <test1> ...

R⁵RS Syntax

The <test> expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then `#t` is returned.

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

or <test1> ...

R⁵RS Syntax

The <test> expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then `#f` is returned.

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

when <test> <expression1> <expression2> ...

STKLOS Syntax

If the <test> expression yields a true value, the <expression>s are evaluated from left to right and the value of the last <expression> is returned. Otherwise, `when` returns *void*.

unless <test> <expression1> <expression2> ...

STKLOS Syntax

If the <test> expression yields a false value, the <expression>s are evaluated from left to right and the value of the last <expression> is returned. Otherwise, `unless` returns *void*.

4.4 Binding Constructs

The three binding constructs `let`, `let*`, and `letrec` are available in STKLOS. These constructs differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially; while in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

let *<bindings>* *<body>* *R⁵RS* Syntax
let *<variable>* *<bindings>* *<body>* *R⁵RS* Syntax

In a `let`, *<bindings>* should have the form

```
((<variable1> <init1>) ...)
```

where each *<init>* is an expression, and *<body>* should be a sequence of one or more expressions. It is an error for a *<variable>* to appear more than once in the list of variables being bound.

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 value(s) of the last expression of *<body>* is(are) returned. Each binding of a *<variable>* has *<body>* as its region.

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

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

The second form of `let`, which is generally called a *named let*, is a variant on the syntax of `let` which provides a more general looping construct than `do` (see [do], page 9) and may also be used to express recursions. It has the same syntax and semantics as ordinary `let` except that *<variable>* is bound within *<body>* to a procedure whose formal arguments are the bound variables and whose body is *<body>*. Thus the execution of *<body>* may be repeated by invoking the procedure named by *<variable>*.

```
(let loop ((numbers '(3 -2 1 6 -5))
          (nonneg '())
          (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg))))))
⇒ ((6 1 3) (-5 -2))
```

let* *<bindings>* *<body>* *R⁵RS* Syntax

In a `let*`, *<bindings>* should have the same form as in a `let` (however, a *<variable>* can appear more than once in the list of variables being bound).

`Let*` is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a binding indicated by

```
(<variable> <init>)
```

is that part of the `let*` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

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

letrec <bindings> <body>

R⁵RS Syntax

<bindings> should have the form as in `let`.

The <variable>s are bound to fresh locations holding undefined values, 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 value(s) of the last expression in <body> is(are) returned. Each binding of a <variable> has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

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

fluid-let <bindings> <body>

STKLOS Syntax

The <bindings> are evaluated in the current environment, in some unspecified order, the current values of the variables present in <bindings> are saved, and the new evaluated values are assigned to the <bindings> variables. Once this is done, the expressions of <body> are evaluated sequentially in the current environment; the value of the last expression is the result of `fluid-let`. Upon exit, the stored variables values are restored. An error is signalled if any of the <bindings> variable is unbound.

```
(let* ((a 'out)
      (f (lambda () a)))
  (list (f)
        (fluid-let ((a 'in)) (f))
        (f))) ⇒ (out in out)
```

When the body of a `fluid-let` is exited by invoking a continuation, the new variable values are saved, and the variables are set to their old values. Then, if the body is reentered by invoking a continuation, the old values are saved and new values are restored. The following example illustrates this behavior

```
(let ((cont #f)
      (l '())
      (a 'out))
  (set! l (cons a l))
  (fluid-let ((a 'in))
    (set! cont (call-with-current-continuation (lambda (k) k)))
    (set! l (cons a l)))
  (set! l (cons a l)))
```

```
(if cont (cont #f) 1)) ⇒ (out in out in out)
```

4.5 Sequencing

begin <expression1> <expression2> ...

R⁵RS Syntax

The <expression>s are evaluated sequentially from left to right, and the value(s) of the last <expression> is(are) returned. This expression type is used to sequence side effects such as input and output.

```
(define x 0)
```

```
(begin (set! x 5)
       (+ x 1)) ⇒ 6
```

```
(begin (display "4 plus 1 equals ")
       (display (+ 4 1))) ↯ 4 plus 1 equals 5
      ⇒ void
```

4.6 Iterations

do [[<var1> <init1> <step1>] ...] [<test> <expr> ...] <command> ...

R⁵RS Syntax

Do 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. When a termination condition is met, the loop exits after evaluating the <expr>s.

Do expressions are evaluated as follows: The <init> expressions are evaluated (in some unspecified order), the <var>s are bound to fresh locations, the results of the <init> expressions are stored in the bindings of the <var>s, and then the iteration phase begins.

Each iteration begins by evaluating <test>; if the result is false then the <command> expressions are evaluated in order for effect, the <step> expressions are evaluated in some unspecified order, the <var>s are bound to fresh locations, the results of the <step>s are stored in the bindings of the <var>s, and the next iteration begins.

If <test> evaluates to a true value, then the <expr>s are evaluated from left to right and the value(s) of the last <expr> is(are) returned. If no <expr>s are present, then the value of the do expression is *void*.

The region of the binding of a <var> consists of the entire do expression except for the <init>s. It is an error for a <var> to appear more than once in the list of do variables.

A <step> may be omitted, in which case the effect is the same as if

```
(<var> <init> <var>)
```

had been written.

```
(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
```

dotimes [*var count*] <expression1> <expression2> ... STKLOS Syntax

dotimes [*var count result*] <expression1> <expression2> ... STKLOS Syntax

Evaluates the `count` expression, which must return an integer and then evaluates the <expression>s once for each integer from zero (inclusive) to `count` (exclusive), in order, with the symbol `var` bound to the integer; if the value of `count` is zero or negative, then the <expression>s are not evaluated. When the loop completes, `result` is evaluated and its value is returned as the value of the `dotimes` construction. If `result` is omitted, `dotimes` result is *void*.

```
(let ((l '()))
  (dotimes (i 4 1)
    (set! l (cons i l)))) ⇒ (3 2 1 0)
```

while <test> <expression1> <expression2> ... STKLOS Syntax

While evaluates the <expression>s until <test> returns a false value. The value returned by this form is *void*

until <test> <expression1> <expression2> ... STKLOS Syntax

until evaluates the <expression>s until <while> returns a false value. The value returned by this form is *void*

4.7 Delayed Evaluation

delay <expression> *R⁵RS* Procedure

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 unpredictable.

See the description of `force` (see [force], page 39) for a more complete description of `delay`.

promise? *obj* STKLOS Procedure

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

4.8 Quasiquotation

quasiquote <template> *R⁵RS* Syntax
 ‘<template>’ *R⁵RS* Syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the <template>, the result of evaluating ‘<template>’ is equivalent to the result of evaluating ‘<template>’. If a comma appears within the <template>, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence. A comma at-sign should only appear within a list or vector <template>.

```

'(list ,(+ 1 2) 4) ⇒ (list 3 4)
(let ((name 'a)) '(list ,name ,name))
                    ⇒ (list a (quote a))
'(a ,(+ 1 2) ,(map abs '(4 -5 6)) b)
                    ⇒ (a 3 4 5 6 b)
'((foo ,(- 10 3)) ,(cdr '(c)) . ,(car '(cons)))
                    ⇒ ((foo 7) . cons)
'#(10 5 ,(sqrt 4) ,(map sqrt '(16 9)) 8)
                    ⇒ #(10 5 2 4 3 8)

```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquote, and decreases by one inside each unquote.

```

'(a '(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
    ⇒ (a '(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  '(a '(b ,,name1 ,',name2 d) e))
    ⇒ (a '(b ,x ,',y d) e)

```

The two notations `'<template>` and `(quasiquote <template>)` are identical in all respects. `,<expression>` is identical to `(unquote <expression>)`, and `,@<expression>` is identical to `(unquote-splicing <expression>)`.

4.9 Macros

STKLOS supports hygienic macros such as the ones defined in *R⁵RS* as well as low level macros. Low level macros are defined with `define-macro` whereas *R⁵RS* macros are defined with `define-syntax`¹. Hygienic macros use the implementation of "Macro by Example" (Eugene Kohlbecker, *R⁴RS*) by Dorai Sitaram. This implementation generates low level STKLOS macros. This implementation of hygienic macros is not expensive.

Its major drawback is that these macros are not *referentially transparent* (see section 'Macros' in *R⁴RS*). Lexically scoped macros (i.e., `let-syntax` and `letrec-syntax`) are not supported. In any case, the problem of referential transparency gains poignancy only when `let-syntax` and `letrec-syntax` are used. So you will not be courting large-scale disaster unless you're using system-function names as local variables with unintuitive bindings that the macro can't use. However, if you must have the full *R⁵RS* macro functionality, you can load do

```
(require "full-syntax")
```

to have access to the more featureful (but also more expensive) versions of `syntax-rules`. Requiring `full-syntax` load the version 2.1 of an implementation of hygienic macros by Robert Hieb and R. Kent Dybvig.

TODO: DEFINE THE LOW LEVEL EXPANDER MECHANISM

define-macro [*<name>* *<formals>*] *<body>* STKLOS Syntax

define-macro *<name>* [*lambda* *<formals>*] *<body>* STKLOS Syntax

`define-macro` can be used to define low-level macro (i.e. *non hygienic* macros. This for is similar to the `defmacro` of Common Lisp.

```
(define-macro (incr x) '(set! ,x (+ ,x 1)))
(let ((a 1)) (incr a) a) ⇒ 2
```

```
(define-macro (when test . body)
```

¹ documentation about hygienic macros has been stolen in the SLIB manual

```

      '(if ,test ,@(if (null? (cdr body)) body '((begin ,@body))))
      (macro-expand '(when a b)) ⇒ (if a b)
      (macro-expand '(when a b c d))
        ⇒ (if a (begin b c d))

      (define-macro (my-and . exprs)
        (cond
          ((null? exprs)          #t)
          ((= (length exprs) 1) (car exprs))
          (else                   '(if ,(car exprs)
            (my-and ,@(cdr exprs))
            #f))))
      (macro-expand '(my-and a b c))
        ⇒ (if a (my-and b c) #f)

```

define-syntax *<identifier>* *<transformer-spec>* *R⁵RS Syntax*
<Define-syntax> extends the top-level syntactic environment by binding the *<identifier>* to the specified transformer.

Note: *<transformer-spec>* should be an instance of *syntax-rules*.

```

      (define-syntax let*
        (syntax-rules ()
          ((let* () body1 body2 ...)
            (let () body1 body2 ...))
          ((let* ((name1 val1) (name2 val2) ...)
            body1 body2 ...)
            (let ((name1 val1))
              (let* (( name2 val2) ...)
                body1 body2 ...))))

```

syntax-rules *< literals >* *< syntax-rule >* ... *R⁵RS Syntax*
< literals > is a list of identifiers, and each *< syntax-rule >* should be of the form

```
(pattern template)
```

An instance of *< syntax-rules >* produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose name is associated with a transformer specified by *< syntax-rules >* is matched against the patterns contained in the *< syntax-rules >*, beginning with the leftmost *syntax-rule*. When a match is found, the macro use is transcribed hygienically according to the template.

Each pattern begins with the name for the macro. This name is not involved in the matching and is not considered a pattern variable or literal identifier.

Note: For a complete description of the Scheme pattern language, refer to *R⁵RS* .

let-syntax *< bindings >* *< body >* *R⁵RS Syntax*
< Bindings > should have the form

```
((< keyword > < transformer spec >) ...)
```

Each *< keyword >* is an identifier, each *< transformer spec >* is an instance of *syntax-rules*, and *< body >* should be a sequence of one or more expressions. It is an error for a *< keyword >* to appear more than once in the list of keywords being bound.

The *< body >* is expanded in the syntactic environment obtained by extending the syntactic environment of the *let-syntax* expression with macros whose keywords are the

<keyword>s, bound to the specified transformers. Each binding of a <keyword> has <body> as its region.

Note: `let-syntax` is available only after having required the file "full-syntax".

```
(let-syntax ((when (syntax-rules ()
  ((when test stmt1 stmt2 ...)
    (if test
      (begin stmt1
        stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now))
    if))                                     ⇒ now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m))))                                 ⇒ outer
```

letrec-syntax <bindings> <body>

R⁵RS Syntax

Syntax of `letrec-syntax` is the same as for `let-syntax`.

The <body> is expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` expression 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 <body> within its region, so the transformers can transcribe expressions into uses of the macros introduced by the `letrec-syntax` expression. **Note:** `letrec-syntax` is available only after having required the file "full-syntax".

```
(letrec-syntax
  ((my-or (syntax-rules ()
    ((my-or) #f)
    ((my-or e) e)
    ((my-or e1 e2 ...)
      (let ((temp e1))
        (if temp
          temp
          (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
      (let temp)
      (if y)
      y)))                                     ⇒ 7
```

macro-expand *form*

STKLOS Procedure

Returns the macro expansion of `form` if it is a macro call, otherwise `form` is returned unchanged.

```
(define-macro (incr x) '(set! ,x (+ ,x 1)))
(macro-expand '(incr foo)) ⇒ (set! foo (+ foo 1))
(macro-expand '(car bar)) ⇒ (car bar)
```

5 Program structure

R⁵RS discusses how to structure programs. Everything which is defined in Section 5 of *R⁵RS* applies also to STKLOS. To make things shorter, this aspects will not be described here (see *R⁵RS* for complete information).

STKLOS modules can be used to organize a program into separate environments (or *name spaces*). Modules provide a clean way to organize and enforce the barriers between the components of a program.

STKLOS provides a simple module system which is largely inspired from the one of Tung and Dybvig exposed in *Tung-Dybvig-96*. As their modules system, STKLOS modules are defined to be easily used in an interactive environment.

define-module *<name>* *<expr1>* *<expr2>* ... STKLOS Syntax

Define-module evaluates the expressions *<expr1>*, *<expr2>* ... which constitute the body of the module *<name>* in the environment of that module. *Name* must be a valid symbol. If this symbol has not already been used to define a module, a new module, named *name*, is created. Otherwise, the expressions *<expr1>*, *<expr2>* ... are evaluated in the environment of the (old) module *<name>*². Definitions done in a module are local to the module and do not interact with the definitions in other modules. Consider the following definitions,

```
(define-module M1
  (define a 1))

(define-module M2
  (define a 2)
  (define b (* 2 x)))
```

Here, two modules are defined and they both bind the symbol *a* to a value. However, since *a* has been defined in two distinct modules they denote two different locations.

The *STklos* module, which is predefined, is a special module which contains all the *global variables* of a *R⁵RS* program. A symbol defined in the *STklos* module, if not hidden by a local definition, is always visible from inside a module. So, in the previous exemple, the *x* symbol refers the *x* symbol defined in the *STklos* module.

The result of **define-module** is *void*.

current-module STKLOS Procedure

Returns the current module.

```
(define-module M
  (display
    (cons (eq? (current-module) (find-module 'M))
          (eq? (current-module) (find-module 'STklos))))))
→ (#t . #f)
```

find-module *name* STKLOS Procedure

find-module *name default* STKLOS Procedure

STKLOS modules are first class objects and **find-module** returns the module associated to *name* if it exists. If there is no module associated to *name*, an error is signaled if no *default* is provided, otherwise **find-module** returns *default*.

² In fact **define-module** on a given name defines a new module only the first time it is invoked on this name. By this way, interactively reloading a module does not define a new entity, and the other modules which use it are not altered.

module? *object*

STKLOS Procedure

Returns #t if *object* is a module and #f otherwise.

```
(module? (find-module 'STklos)) ⇒ #t
(module? 'STklos)                ⇒ #f
(module? 1 'no)                  ⇒ no
```

export <*symbol1*> <*symbol2*> ...

STKLOS Syntax

Specifies the symbols which are exported (i.e. *visible*) outside the current module. By default, symbols defined in a module are not visible outside this module, excepted if they appear in an **export** clause.

If several **export** clauses appear in a module, the set of exported symbols is determined by “*unionizing*” symbols exported in all the **export** clauses.

The result of **export** is *void*.

import <*module1*> <*module2*> ...

STKLOS Syntax

Specifies the modules which are imported by the current module. Importing a module makes the symbols it exports visible to the importer, if not hidden by local definitions. When a symbol is exported by several of the imported modules, the location denoted by this symbol in the importer module correspond to the one of the first module in the list

```
(<module1> <module2> ...)
```

which exports it.

If several **import** clauses appear in a module, the set of imported modules is determined by appending the various list of modules in their apparition order.

```
(define-module M1
  (export a b)
  (define a 'M1-a)
  (define b 'M1-b))

(define-module M2
  (export b c)
  (define b 'M2-b)
  (define c 'M2-c))

(define-module M3
  (import M1 M2)
  (display (list a b c)))  ⇨ (m1-a m1-b m2-c)
```

Note: Importations are not *transitive*: when the module *C* imports the module *B* which is an importer of *A*, the symbols of *A* are not visible from *C*, except by explicitly importing the *A* module from *C*.

Note: The module *STklos*, which contains the *global variables* is always implicitly imported from a module. Furthermore, this module is always placed at the end of the list of imported modules.

select-module <*name*>

STKLOS Syntax

Changes the value of the current module to the module with the given **name**. The expressions evaluated after **select-module** will take place in module **name** environment. Module **name** must have been created previously by a **define-module**. The result of **select-module** is *void*. **Select-module** is particularly useful when debugging since it

allows to place toplevel evaluation in a particular module. The following transcript shows an usage of `select-module`.³:

```
stklos> (define foo 1)
stklos> (define-module bar
         (define foo 2))
stklos> foo
1
stklos> (select-module bar)
bar> foo
2
bar> (select-module stklos)
stklos>
```

symbol-value *symbol module* STKLOS Procedure

symbol-value *symbol module default* STKLOS Procedure

Returns the value bound to **symbol** in **module**. If **symbol** is not bound, an error is signaled if no **default** is provided, otherwise **symbol-value** returns **default**.

symbol-value* *symbol module* STKLOS Procedure

symbol-value* *symbol module default* STKLOS Procedure

Returns the value bound to **symbol** in **module**. If **symbol** is not bound, an error is signaled if no **default** is provided, otherwise **symbol-value** returns **default**.

Note that this function searches the value of **symbol** in **module** **and** all the modules it imports whereas **symbol-value** searches only in **module**.

module-name *module* STKLOS Procedure

Returns the the name (a symbol) associated to a **module**.

module-imports *module* STKLOS Procedure

Returns the list of modules that **module** imports.

module-exports *module* STKLOS Procedure

Returns the list of symbols exported by **module**. Note that this function returns the list of symbols given in the module **export** clause and that some of these symbols can be not yet defined.

module-symbols *module* STKLOS Procedure

Returns the list of symbols already defined in **module**.

all-modules STKLOS Procedure

Returns the list of all the living modules.

in-module *mod s* STKLOS Syntax

in-module *mod s default* STKLOS Syntax

This form returns the value of **symbol** with name **s** in the module with name **mod**. If this symbol is not bound, an error is signaled if no **default** is provided, otherwise **in-module** returns **default**. Note that the value of **s** is searched in **mod** and all the modules it imports.

This form is in fact a shortcut. In effect,

³ This transcript uses the default value for the function `repl-display-prompt` (see see [\[repl-display-prompt\]](#), page 75) which displays the name of the current module in the evaluator prompt.

```
(in-module my-module foo)
is equivalent to
(symbol-value* 'foo (find-module 'my-module))
```

6 Standard Procedures

6.1 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, `eqv?` is the finest or most discriminating, and `equal?` is the coarsest. `Eqv?` is slightly less discriminating than `eqv?`.

eqv? *obj1 obj2* *R⁵RS Procedure*

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if `obj1` and `obj2` should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of `eqv?` holds for all implementations of Scheme.

The `eqv?` procedure returns `#t` if:

```
obj1 and obj2 are both #t or both #f.
obj1 and obj2 are both symbols and
  (string=? (symbol->string obj1)
            (symbol->string obj2))
  ⇒ #t
```

Note: This assumes that neither `obj1` nor `obj2` is an "uninterned symbol".

```
obj1 and obj2 are both keywords and
  (string=? (keyword->string obj1)
            (keyword->string obj2))
  ⇒ #t
```

`obj1` and `obj2` are both numbers, are numerically equal (see see [\[=\]](#), page 19), and are either both exact or both inexact.

`obj1` and `obj2` are both characters and are the same character according to the `char=?` procedure (see see [\[char=\]](#), page 31).

both `obj1` and `obj2` are the empty list.

`obj1` and `obj2` are pairs, vectors, or strings that denote the same locations in the store.

`obj1` and `obj2` are procedures whose location tags are equal.

Note: STKLOS extends *R⁵RS* `eqv?` to take into account the keyword type.

Here are some examples:

```
(eqv? 'a 'a)           ⇒ #t
(eqv? 'a 'b)           ⇒ #f
(eqv? 2 2)             ⇒ #t
(eqv? :foo :foo)       ⇒ #t
(eqv? :foo :bar)       ⇒ #f
(eqv? '() '())         ⇒ #t
(eqv? 100000000 100000000) ⇒ #t
```

```

(eqv? (cons 1 2) (cons 1 2))    ⇒ #f
(eqv? (lambda () 1)
      (lambda () 2))          ⇒ #f
(eqv? #f 'nil)                ⇒ #f
(let ((p (lambda (x) x)))
      (eqv? p p))              ⇒ #t

```

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

```

(eqv? "" "")                  ⇒ unspecified
(eqv? '#() '#())              ⇒ unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))          ⇒ unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))          ⇒ unspecified

```

Note: In fact, the value returned by STKLOS depends of the way code is entered and can yield `#t` in some cases and `#f` in others.

See *R⁵RS* for more details on `eqv?`

`eq?` *obj1 obj2*

R⁵RS Procedure

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

`Eq?` and `eqv?` are guaranteed to have the same behavior on symbols, keywords, booleans, the empty list, pairs, procedures, and non-empty strings and vectors. `Eq?`'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when `eqv?` would also return true. `Eq?` may also behave differently from `eqv?` on empty vectors and empty strings.

Note: STKLOS extends *R⁵RS* `eq?` to take into account the keyword type.

Note: In STKLOS, comparison of character returns `#t` for identical characters and `#f` otherwise.

```

(eq? 'a 'a)                   ⇒ #t
(eq? '(a) '(a))                ⇒ unspecified
(eq? (list 'a) (list 'a))      ⇒ #f
(eq? "a" "a")                  ⇒ unspecified
(eq? "" "")                     ⇒ unspecified
(eq? :foo :foo)                 ⇒ #t
(eq? :foo :bar)                 ⇒ #f
(eq? '() '())                   ⇒ #t
(eq? 2 2)                       ⇒ unspecified
(eq? #\A #\A)                   ⇒ #t (unspecified in R5RS)
(eq? car car)                   ⇒ #t
(let ((n (+ 2 3)))
      (eq? n n))                 ⇒ #t (unspecified in R5RS)
(let ((x '(a)))
      (eq? x x))                 ⇒ #t
(let ((x '#()))
      (eq? x x))                 ⇒ #t
(let ((p (lambda (x) x)))
      (eq? p p))                 ⇒ #t
(eq? :foo :foo)                 ⇒ #t

```

```
(eq? :bar bar:) ⇒ #t
(eq? :bar :foo) ⇒ #f
```

equal? *obj1 obj2**R⁵RS Procedure*

Equal? recursively compares the contents of pairs, vectors, and strings, applying **eq?** on other objects such as numbers and symbols. A rule of thumb is that objects are generally **equal?** if they print the same. **Equal?** may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a) ⇒ #t
(equal? '(a) '(a)) ⇒ #t
(equal? '(a (b) c)
 '(a (b) c)) ⇒ #t
(equal? "abc" "abc") ⇒ #t
(equal? 2 2) ⇒ #t
(equal? (make-vector 5 'a)
 (make-vector 5 'a)) ⇒ #t
```

6.2 Numbers

R⁵RS description of number is quite long and will not be given here. STKLOS support the full number tower as described in *R⁵RS*; see this document for a complete description.

number? *obj**R⁵RS Procedure***complex?** *obj**R⁵RS Procedure***real?** *obj**R⁵RS Procedure***rational?** *obj**R⁵RS Procedure***integer?** *obj**R⁵RS Procedure*

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

If *z* is an inexact complex number, then **(real? z)** is true if and only if **(zero? (imag-part z))** is true. If *x* is an inexact real number, then **(integer? x)** is true if and only if **(= x (round x))**. The **(%bignum? x)** is true iff *x* is represented as a bignum in memory.

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

exact? *z**R⁵RS Procedure***inexact?** *z**R⁵RS Procedure*

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

<code>= z1 z2 z3 ...</code>	<i>R⁵RS</i> Procedure
<code>< x1 x2 x3 ...</code>	<i>R⁵RS</i> Procedure
<code>> x1 x2 x3 ...</code>	<i>R⁵RS</i> Procedure
<code><= x1 x2 x3 ...</code>	<i>R⁵RS</i> Procedure
<code>>= x1 x2 x3 ...</code>	<i>R⁵RS</i> Procedure

These procedures return **#t** if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically non-increasing.

<code>zero? z</code>	<i>R⁵RS</i> Procedure
<code>positive? x</code>	<i>R⁵RS</i> Procedure
<code>negative? x</code>	<i>R⁵RS</i> Procedure
<code>odd? n</code>	<i>R⁵RS</i> Procedure
<code>even? n</code>	<i>R⁵RS</i> Procedure

These numerical predicates test a number for a particular property, returning **#t** or **#f**.

<code>max x1 x2 ...</code>	<i>R⁵RS</i> Procedure
<code>min x1 x2 ...</code>	<i>R⁵RS</i> Procedure

These procedures return the maximum or minimum of their arguments.

<code>(max 3 4)</code>	\Rightarrow	<code>4</code>	; <i>exact</i>
<code>(max 3.9 4)</code>	\Rightarrow	<code>4.0</code>	; <i>inexact</i>

Note: If any argument is inexact, then the result will also be inexact

<code>+ z1 ...</code>	<i>R⁵RS</i> Procedure
<code>* z1 ...</code>	<i>R⁵RS</i> Procedure

These procedures return the sum or product of their arguments.

<code>(+ 3 4)</code>	\Rightarrow	<code>7</code>
<code>(+ 3)</code>	\Rightarrow	<code>3</code>
<code>(+)</code>	\Rightarrow	<code>0</code>
<code>(* 4)</code>	\Rightarrow	<code>4</code>
<code>(*)</code>	\Rightarrow	<code>1</code>

<code>- z</code>	<i>R⁵RS</i> Procedure
<code>- z1 z2</code>	<i>R⁵RS</i> Procedure
<code>/ z</code>	<i>R⁵RS</i> Procedure
<code>/ z1 z2 ...</code>	<i>R⁵RS</i> Procedure

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

<code>(- 3 4)</code>	\Rightarrow	<code>-1</code>
<code>(- 3 4 5)</code>	\Rightarrow	<code>-6</code>
<code>(- 3)</code>	\Rightarrow	<code>-3</code>
<code>(/ 3 4 5)</code>	\Rightarrow	<code>3/20</code>
<code>(/ 3)</code>	\Rightarrow	<code>1/3</code>

<code>abs x</code>	<i>R⁵RS</i> Procedure
--------------------	----------------------------------

Abs returns the absolute value of its argument.

<code>(abs -7)</code>	\Rightarrow	<code>7</code>
-----------------------	---------------	----------------

quotient *n1 n2**R⁵RS* Procedure**remainder** *n1 n2**R⁵RS* Procedure**modulo** *n1 n2**R⁵RS* Procedure

These procedures implement number-theoretic (integer) division. *n2* should be non-zero.

All three procedures return integers. If *n1/n2* is an integer:

```
(quotient n1 n2) ⇒ n1/n2
(remainder n1 n2) ⇒ 0
(modulo n1 n2) ⇒ 0
```

If *n1/n2* is not an integer:

```
(quotient n1 n2) ⇒ nq
(remainder n1 n2) ⇒ nr
(modulo n1 n2) ⇒ nm
```

where *nq* is *n1/n2* rounded towards zero, $0 < \text{abs}(nr) < \text{abs}(n2)$, $0 < \text{abs}(nm) < \text{abs}(n2)$, *nr* and *nm* differ from *n1* by a multiple of *n2*, *nr* has the same sign as *n1*, and *nm* has the same sign as *n2*.

From this we can conclude that for integers *n1* and *n2* with *n2* not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
          (remainder n1 n2))) ⇒ #t
```

provided all numbers involved in that computation are exact.

```
(modulo 13 4) ⇒ 1
(remainder 13 4) ⇒ 1

(modulo -13 4) ⇒ 3
(remainder -13 4) ⇒ -1

(modulo 13 -4) ⇒ -3
(remainder 13 -4) ⇒ 1

(modulo -13 -4) ⇒ -1
(remainder -13 -4) ⇒ -1

(remainder -13 -4.0) ⇒ -1.0 ; inexact
```

gcd *n1 ...**R⁵RS* Procedure**lcm** *n1 ...**R⁵RS* Procedure

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

```
(gcd 32 -36) ⇒ 4
(gcd) ⇒ 0
(lcm 32 -36) ⇒ 288
(lcm 32.0 -36) ⇒ 288.0 ; inexact
(lcm) ⇒ 1
```

numerator *q**R⁵RS* Procedure**denominator** *q**R⁵RS* Procedure

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

```
(numerator (/ 6 4)) ⇒ 3
(denominator (/ 6 4)) ⇒ 2
```

```
(denominator
 (exact->inexact (/ 6 4))) => 2.0
```

floor <i>x</i>	<i>R⁵RS</i> Procedure
ceiling <i>x</i>	<i>R⁵RS</i> Procedure
truncate <i>x</i>	<i>R⁵RS</i> Procedure
round <i>x</i>	<i>R⁵RS</i> Procedure

These procedures return integers. **Floor** returns the largest integer not larger than *x*. **Ceiling** returns the smallest integer not smaller than *x*. **Truncate** returns the integer closest to *x* whose absolute value is not larger than the absolute value of *x*. **Round** returns the closest integer to *x*, rounding to even when *x* is halfway between two integers.

Rationale: **Round** rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

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

```
(floor -4.3)      => -5.0
(ceiling -4.3)   => -4.0
(truncate -4.3)  => -4.0
(round -4.3)     => -4.0

(floor 3.5)      => 3.0
(ceiling 3.5)   => 4.0
(truncate 3.5)  => 3.0
(round 3.5)     => 4.0 ; inexact

(round 7/2)      => 4    ; exact
(round 7)        => 7
```

rationalize <i>x y</i>	<i>R⁵RS</i> Procedure
-------------------------------	----------------------------------

Rationalize returns the simplest rational number differing from *x* by no more than *y*. A rational number *r1* is simpler than another rational number *r2* if *r1* = *p1/q1* and *r2* = *p2/q2* (in lowest terms) and *abs(p1)* <= *abs(p2)* and *abs(q1)* <= *abs(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.

```
(rationalize
 (inexact->exact .3) 1/10) => 1/3    ; exact
(rationalize .3 1/10)  => #i1/3   ; inexact
```

exp <i>z</i>	<i>R⁵RS</i> Procedure
log <i>z</i>	<i>R⁵RS</i> Procedure
sin <i>z</i>	<i>R⁵RS</i> Procedure
cos <i>z</i>	<i>R⁵RS</i> Procedure
tan <i>z</i>	<i>R⁵RS</i> Procedure
asin <i>z</i>	<i>R⁵RS</i> Procedure
acos <i>z</i>	<i>R⁵RS</i> Procedure
atan <i>z</i>	<i>R⁵RS</i> Procedure
atan <i>y x</i>	<i>R⁵RS</i> Procedure

These procedures compute the usual transcendental functions. **Log** computes the natural logarithm of *z* (not the base ten logarithm). **Asin**, **acos**, and **atan** compute arcsine, arccosine, and arctangent, respectively. The two-argument variant of **atan** computes

(**angle** (**make-rectangular** *x y*))

When it is possible these procedures produce a real result from a real argument.

sqrt <i>z</i>	<i>R⁵RS</i> Procedure
----------------------	----------------------------------

Returns the principal square root of *z*. The result will have either positive real part, or zero real part and non-negative imaginary part.

expt <i>z1 z2</i>	<i>R⁵RS</i> Procedure
--------------------------	----------------------------------

Returns *z1* raised to the power *z2*.
Note: 0^z is 1 if $z = 0$ and 0 otherwise.

make-rectangular <i>x1 x2</i>	<i>R⁵RS</i> Procedure
make-polar <i>x3 x4</i>	<i>R⁵RS</i> Procedure
real-part <i>z</i>	<i>R⁵RS</i> Procedure
imag-part <i>z</i>	<i>R⁵RS</i> Procedure
magnitude <i>z</i>	<i>R⁵RS</i> Procedure
angle <i>z</i>	<i>R⁵RS</i> Procedure

If *x1*, *x2*, *x3*, and *x4* are real numbers and *z* is a complex number such that

$$z = x1 + x2.i = x3.e^{i.x4}$$

Then

(make-rectangular <i>x1 x2</i>)	$\Rightarrow z$
(make-polar <i>x3 x4</i>)	$\Rightarrow z$
(real-part <i>z</i>)	$\Rightarrow x1$
(imag-part <i>z</i>)	$\Rightarrow x2$
(magnitude <i>z</i>)	$\Rightarrow \text{abs}(x3)$
(angle <i>z</i>)	$\Rightarrow xa$

where $-\pi < xa \leq \pi$ with $xa = x4 + 2\pi n$ for some integer *n*.

Note: **Magnitude** is the same as **abs** for a real argument.

exact->inexact <i>z</i>	<i>R⁵RS</i> Procedure
inexact->exact <i>z</i>	<i>R⁵RS</i> Procedure

Exact->inexact returns an inexact representation of *z*. The value returned is the inexact number that is numerically closest to the argument. **Inexact->exact** returns an exact representation of *z*. The value returned is the exact number that is numerically closest to the argument.

number->string <i>z</i>	<i>R⁵RS</i> Procedure
number->string <i>z radix</i>	<i>R⁵RS</i> Procedure

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, **radix** defaults to 10. The procedure **number->string** takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

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

is true. It is an error if no possible result makes this expression true.

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

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

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

Rationale: If *z* is an inexact number represented using flonums, and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and non-flonum representations.

string->number *string* *R*⁵*RS* Procedure
string->number *string radix* *R*⁵*RS* Procedure

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

```
(string->number "100")      ⇒ 100
(string->number "100" 16)   ⇒ 256
(string->number "1e2")      ⇒ 100.0
(string->number "15##")     ⇒ 1500.0
```

bit-and *n1 n2 ...* STKLOS Procedure
bit-or *n1 n2 ...* STKLOS Procedure
bit-xor *n1 n2 ...* STKLOS Procedure
bit-not *n* STKLOS Procedure
bit-shift *n m* STKLOS Procedure

These procedures allow the manipulation of integers as bit fields. The integers can be of arbitrary length. `bit-and`, `bit-or` and `bit-xor` respectively compute the bitwise *and*, inclusive and exclusive *or*. `bit-not` returns the bitwise *not* of *n*. `bit-shift` returns the bitwise *shift* of *n*. The integer *n* is shifted left by *m* bits; If *m* is negative, *n* is shifted right by $-m$ bits.

```
(bit-or 5 3)      ⇒ 7
(bit-xor 5 3)     ⇒ 6
(bit-and 5 3)     ⇒ 1
(bit-not 5)       ⇒ -6
(bit-or 1 2 4 8) ⇒ 15
(bit-shift 5 3)   ⇒ 40
(bit-shift 5 -1)  ⇒ 2
```

random-integer *n* STKLOS Procedure

Return an integer in the range $[0, \dots, n]$. Subsequent results of this procedure appear to be independent uniformly distributed over the range $[0, \dots, n]$. The argument *n* must be a positive integer, otherwise an error is signaled. This function is equivalent to the eponym function of SRFI-27 (see SRFI-27 definition for more details)

random-real

STKLOS Procedure

Return a real number *r* such that $0 < r < 1$. Subsequent results of this procedure appear to be independent uniformly distributed. This function is equivalent to the eponym function of SRFI-27 (see SRFI-27 definition for more details)

6.3 Booleans

Of all the standard Scheme values, only **#f** counts as false in conditional expressions. Except for **#f**, all standard Scheme values, including **#t**, pairs, the empty list, symbols, numbers, strings, vectors, and procedures, count as true.

Boolean constants evaluate to themselves, so they do not need to be quoted in programs.

not *obj**R⁵RS* Procedure

Not returns **#t** if *obj* is false, and returns **#f** otherwise.

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

boolean? *obj**R⁵RS* Procedure

Boolean? returns **#t** if *obj* is either **#t** or **#f** and returns **#f** otherwise.

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

6.4 Pairs and lists**pair?** *obj**R⁵RS* Procedure

Pair? returns **#t** if *obj* is a pair, and otherwise returns **#f**.

cons *obj1 obj2**R⁵RS* Procedure

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.

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

car *pair**R⁵RS* Procedure

Returns the contents of the car field of *pair*. Note that it is an error to take the **car** of the empty list.

```
(car '(a b c))      ⇒ a
(car '((a) b c d))  ⇒ (a)
(car '(1 . 2))      ⇒ 1
(car '())           ⇒ error
```

cdr *pair* *R⁵RS Procedure*

Returns the contents of the `cdr` field of `pair`. Note that it is an error to take the `cdr` of the empty list.

```
(cdr '((a) b c d))    ⇒ (b c d)
(cdr '(1 . 2))       ⇒ 2
(cdr '())            ⇒ error
```

set-car! *pair obj* *R⁵RS Procedure*

Stores `obj` in the `car` field of `pair`. The value returned by `set-car!` is *void*.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)
(set-car! (g) 3)      ⇒ error
```

set-cdr! *pair obj* *R⁵RS Procedure*

Stores `obj` in the `cdr` field of `pair`. The value returned by `set-cdr!` is *void*.

caar *pair* *R⁵RS Procedure*

cadr *pair* *R⁵RS Procedure*

⋮

cdddar *pair* *R⁵RS Procedure*

cddddr *pair* *R⁵RS Procedure*

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

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

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

null? *obj* *R⁵RS Procedure*

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

pair-mutable? *obj* STKLOS Procedure

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

```
(pair-mutable? '(1 . 2)) ⇒ #f
(pair-mutable? (cons 1 2)) ⇒ #t
(pair-mutable? 12) ⇒ #f
```

list? *obj* *R⁵RS Procedure*

Returns `#t` if `obj` is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list.

```
(list? '(a b c))    ⇒ #t
(list? '())         ⇒ #t
(list? '(a . b))   ⇒ #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))       ⇒ #f
```

list *obj ...* *R⁵RS Procedure*

Returns a newly allocated list of its arguments.

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

list* *obj ...* STKLOS Procedure

`list*` is like `list` except that the last argument to `list*` is used as the *cdr* of the last pair constructed.

```
(list* 1 2 3)           ⇒ (1 2 . 3)
(list* 1 2 3 '(4 5)) ⇒ (1 2 3 4 5)
(list*)                 ⇒ ()
```

length *list* *R⁵RS* Procedure

Returns the length of `list`.

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

append *list ...* *R⁵RS* Procedure

Returns a list consisting of the elements of the first list followed by the elements of the other lists.

```
(append '(x) '(y))           ⇒ (x y)
(append '(a) '(b c d))       ⇒ (a b c d)
(append '(a (b)) '((c)))    ⇒ (a (b) (c))
```

The resulting list is always newly allocated, except that it shares structure with the last list argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))    ⇒ (a b c . d)
(append '() 'a)             ⇒ a
```

append! *list ...* STKLOS Procedure

Returns a list consisting of the elements of the first list followed by the elements of the other lists. Contrarily to `append`, the parameter lists (except the last one) are physically modified: their last pair is changed to the value of the next list in the `append!` formal parameter list.

```
(let* ((l1 (list 1 2))
       (l2 (list 3))
       (l3 (list 4 5))
       (l4 (append! l1 l2 l3)))
  (list l1 l2 l3)) ⇒ ((1 2 3 4 5) (3 4 5) (4 5))
```

An error is signaled if one of the given lists is a constant list.

reverse *list* *R⁵RS* Procedure

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

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

reverse! *list* STKLOS Procedure

Returns a list consisting of the elements of `list` in reverse order. Contrarily to `reverse`, the returned value is not newly allocated but computed "in place".

```
(let ((l '(a b c)))
  (list (reverse! l) l)) ⇒ ((c b a) (a))
(reverse! '(a constant list)) ⇒ error
```

list-tail *list k**R⁵RS Procedure*

Returns the sublist of *list* obtained by omitting the first *k* elements. It is an error if *list* has fewer than *k* elements. List-tail could be defined by

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

last-pair *list**STKLOS Procedure*

Returns the last pair of *list*.

```
(last-pair '(1 2 3)) ⇒ (3)
(last-pair '(1 2 . 3)) ⇒ (2 . 3)
```

list-ref *list k**R⁵RS Procedure*

Returns the *k*th element of *list*. (This is the same as the car of (list-tail *list* *k*.) It is an error if *list* has fewer than *k* elements.

```
(list-ref '(a b c d) 2) ⇒ c
(list-ref '(a b c d)
  (inexact->exact (round 1.8))) ⇒ c
```

memq *obj list**R⁵RS Procedure***memv** *obj list**R⁵RS Procedure***member** *obj list**R⁵RS Procedure*

These procedures return the first sublist of *list* whose car is *obj*, where the sublists of *list* are the non-empty lists returned by (list-tail *list* *k*) for *k* less than the length of *list*. If *obj* does not occur in *list*, then #f (not the empty list) is returned. Memq uses eq? to compare *obj* with the elements of *list*, while memv uses eqv? and member uses equal?.

```
(memq 'a '(a b c)) ⇒ (a b c)
(memq 'b '(a b c)) ⇒ (b c)
(memq 'a '(b c d)) ⇒ #f
(memq (list 'a) '(b (a) c)) ⇒ #f
(member (list 'a)
  '(b (a) c)) ⇒ ((a) c)
(memv 101 '(100 101 102)) ⇒ (101 102)
```

assq *obj alist**R⁵RS Procedure***assv** *obj alist**R⁵RS Procedure***assoc** *obj alist**R⁵RS Procedure*

Alist (for "association list") must be a list of pairs. These procedures find the first pair in *alist* whose car field is *obj*, and returns that pair. If no pair in *alist* has *obj* as its car, then #f (not the empty list) is returned. Assq uses eq? to compare *obj* with the car fields of the pairs in *alist*, while assv uses eqv? and assoc uses equal?.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e) ⇒ (a 1)
(assq 'b e) ⇒ (b 2)
(assq 'd e) ⇒ #f
(assq (list 'a) '((a)) ((b)) ((c)))) ⇒ #f
(assoc (list 'a) '((a)) ((b)) ((c)))) ⇒ ((a))
```

```
(assv 5 '((2 3) (5 7) (11 13)))
      ⇒ (5 7)
```

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

copy-tree *obj*

STKLOS Procedure

`Copy-tree` recursively copies trees of pairs. If `obj` is not a pair, it is returned; otherwise the result is a new pair whose `car` and `cdr` are obtained by calling `copy-tree` on the `car` and `cdr` of `obj`, respectively.

filter *pred list*

STKLOS Procedure

filter! *pred list*

STKLOS Procedure

`Filter` returns all the elements of `list` that satisfy predicate `pred`. The `list` is not disordered: elements that appear in the result list occur in the same order as they occur in the argument list. `Filter!` does the same job than `filter` by physically modifying its `list` argument

```
(filter even? '(0 7 8 8 43 -4)) ⇒ (0 8 8 -4)
(let* ((l1 (list 0 7 8 8 43 -4))
       (l2 (filter! even? l1)))
  (list l1 l2)) ⇒ ((0 8 8 -4) (0 8 8 -4))
```

An error is signaled if `list` is a constant list.

remove *pred list*

STKLOS Procedure

`Remove` returns `list` without the elements that satisfy predicate `pred`:

```
(lambda (pred list) (filter (lambda (x) (not (pred x))) list))
```

The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. `Remove!` does the same job than `remove` by physically modifying its `list` argument

```
(remove even? '(0 7 8 8 43 -4)) ⇒ (7 43)
```

delete *x list [=]*

STKLOS Procedure

delete! *x list [=]*

STKLOS Procedure

`Delete` uses the comparison procedure `=`, which defaults to `equal?`, to find all elements of `list` that are equal to `x`, and deletes them from `list`. The dynamic order in which the various applications of `=` are made is not specified.

The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list.

The comparison procedure is used in this way: `(= x ei)`. That is, `x` is always the first argument, and a list element is always the second argument. The comparison procedure will be used to compare each element of `list` exactly once; the order in which it is applied to the various `ei` is not specified. Thus, one can reliably remove all the numbers greater than five from a list with

```
(delete 5 list <)
```

`delete!` is the linear-update variant of `delete`. It is allowed, but not required, to alter the cons cells in its argument `list` to construct the result.

6.5 Symbols

The STKLOS reader can read symbols whose names contain special characters or letters in the non standard case. When a symbol is read, the parts enclosed in bars (“|”) will be entered verbatim into the symbol’s name. The “|” characters are not part of the symbol; they only serve to delimit the sequence of characters that must be entered “as is”. In order to maintain read-write invariance, symbols containing such sequences of special characters will be written between a pair of “|”.

```
'|a|           ⇒ a
(string->symbol "a") ⇒ |A|
(symbol->string '|A|) ⇒ "A"
'|a b|        ⇒ |a b|
'a|B|c        ⇒ |aBc|
(write '|Fo0|)  ⇐ |Fo0|
(display '|Fo0|) ⇐ Fo0
```

symbol? *obj*

R⁵RS Procedure

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

```
(symbol? 'foo)           ⇒ #t
(symbol? (car '(a b)))  ⇒ #t
(symbol? "bar")        ⇒ #f
(symbol? 'nil)         ⇒ #t
(symbol? '())          ⇒ #f
(symbol? #f)           ⇒ #f
(symbol? :key)         ⇒ #f
```

string->string *string*

R⁵RS Procedure

Returns the name of *symbol* as a string. If the symbol was part of an object returned as the value of a literal expression or by a call to the **read** procedure, and its name contains alphabetic characters, then the string returned will contain characters in the implementation’s preferred standard case – STKLOS prefers lower case. If the symbol was returned by **string->symbol**, the case of characters in the string returned will be the same as the case in the string that was passed to **string->symbol**. It is an error to apply mutation procedures like **string-set!** to strings returned by this procedure.

```
(symbol->string 'flying-fish) ⇒ "flying-fish"
(symbol->string 'Martin)     ⇒ "martin"
(symbol->string (string->symbol "Malvina"))
                              ⇒ "Malvina"
```

string->symbol *string*

R⁵RS Procedure

Returns the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves.

```
(eq? 'mISSISSIppi 'mississippi) ⇒ #t
(string->symbol "mISSISSIppi")   ⇒ |mISSISSIppi|
(eq? 'bitBlt (string->symbol "bitBlt"))
                                  ⇒ #f
(eq? 'JollyWog
 (string->symbol
 (symbol->string 'JollyWog))) ⇒ #t
```

```
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D.")))
⇒ #t
```

string->uninterned-symbol *string* STKLOS Procedure

Returns the symbol whose print name is made from the characters of *string*. This symbol is guaranteed to be *unique* (i.e. not `eq?` to any other symbol):

```
(let ((ua (string->uninterned-symbol "a")))
  (list (eq? 'a ua)
        (eqv? 'a ua)
        (eq? ua (string->uninterned-symbol "a"))
        (eqv? ua (string->uninterned-symbol "a"))))
⇒ (#f #t #f #t)
```

gensym STKLOS Procedure

gensym *prefix* STKLOS Procedure

Creates a new symbol. The print name of the generated symbol consists of a prefix (which defaults to `G`) followed by the decimal representation of a number. If *prefix* is specified, it must be either a string or a symbol.

```
(gensym)      ⇒ |G100|
(gensym "foo-") ⇒ foo-101
(gensym 'foo)  ⇒ foo-102
```

6.6 Characters

The following table gives the list of allowed character names with their ASCII equivalent expressed in octal.

<i>name</i>	<i>value</i>	<i>alternate name</i>	<i>name</i>	<i>value</i>	<i>alternate name</i>
nul	000	null	soh	001	
stx	002		etx	003	
eot	004		enq	005	
ack	006		bel	007	bell
bs	010	backspace	ht	011	tab
nl	012	newline	vt	013	
np	014	page	cr	015	return
so	016		si	017	
dle	020		dc1	021	
dc2	022		dc3	023	
dc4	024		nak	025	
syn	026		etb	027	
can	030		em	031	
sub	032		esc	033	escape
fs	034		gs	035	
rs	036		us	037	

char? *obj* *R⁵RS* Procedure

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

char=? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char<? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char>? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char<=? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char>=? <i>char1 char2</i>	<i>R⁵RS</i> Procedure

These procedures impose a total ordering on the set of characters. It is guaranteed that under this ordering:

- The upper case characters are in order.
- The lower case characters are in order.
- The digits are in order.
- Either all the digits precede all the upper case letters, or vice versa.
- Either all the digits precede all the lower case letters, or vice versa.

char-ci=? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char-ci<? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char-ci>? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char-ci<=? <i>char1 char2</i>	<i>R⁵RS</i> Procedure
char-ci>=? <i>char1 char2</i>	<i>R⁵RS</i> Procedure

These procedures are similar to **char=?** et cetera, but they treat upper case and lower case letters as the same. For example, (**char-ci=?** #\A #\a) returns #t.

char-alphabetic? <i>char</i>	<i>R⁵RS</i> Procedure
char-numeric? <i>char</i>	<i>R⁵RS</i> Procedure
char-whitespace? <i>char</i>	<i>R⁵RS</i> Procedure
char-upper-case? <i>letter</i>	<i>R⁵RS</i> Procedure
char-lower-case? <i>letter</i>	<i>R⁵RS</i> Procedure

These procedures return #t if their arguments are alphabetic, numeric, whitespace, upper case, or lower case characters, respectively, otherwise they return #f. The following remarks, which are specific to the ASCII character set, are intended only as a guide: The alphabetic characters are the 52 upper and lower case letters. The numeric characters are the ten decimal digits. The whitespace characters are space, tab, line feed, form feed, and carriage return.

char->integer <i>char</i>	<i>R⁵RS</i> Procedure
integer->char <i>n</i>	<i>R⁵RS</i> Procedure

Given a character, **char->integer** returns an exact integer representation of the character. Given an exact integer that is the image of a character under **char->integer**, **integer->char** returns that character. These procedures implement order-preserving isomorphisms between the set of characters under the **char<=?** ordering and some subset of the integers under the <= ordering. That is, if

$$(\text{char}<=? \text{ a b}) \Rightarrow \#t \quad \text{and} \quad (<= \text{ x y}) \Rightarrow \#t$$

and x and y are in the domain of **integer->char**, then

$$\begin{aligned} (<= (\text{char->integer a}) \\ (\text{char->integer b})) &\Rightarrow \#t \end{aligned}$$

$$\begin{aligned} (\text{char}<=? (\text{integer->char x}) \\ (\text{integer->char y})) &\Rightarrow \#t \end{aligned}$$

char-upcase <i>char</i>	<i>R⁵RS</i> Procedure
char-downcase <i>char</i>	<i>R⁵RS</i> Procedure

These procedures return a character **char2** such that (**char-ci=?** **char char2**). In addition, if **char** is alphabetic, then the result of **char-upcase** is upper case and the result of **char-downcase** is lower case.

6.7 Strings

STKLOS string constants allow the insertion of arbitrary characters by encoding them as escape sequences. An escape sequence is introduced by a backslash “\”. The valid escape sequences are shown in the following table.

<i>Sequence</i>	<i>Character inserted</i>
<code>\b</code>	Backspace
<code>\e</code>	Escape
<code>\n</code>	Newline
<code>\t</code>	Horizontal Tab
<code>\r</code>	Carriage Return
<code>\0abc</code>	ASCII character with octal value abc
<code>\xab</code>	ASCII character with hexadecimal value ab
<code>\<newline></code>	None (permits to enter a string on several lines)
<code>\<other></code>	<other>

For instance, the string

```
"ab\040c\nd\
e"
```

is the string consisting of the characters `#\a`, `#\b`, `#\space`, `#\c`, `#\newline`, `#\d` and `#\e`.

string? *obj* *R⁵RS Procedure*
Returns `#t` if *obj* is a string, otherwise returns `#f`.

make-string *k* *R⁵RS Procedure*
make-string *k char* *R⁵RS Procedure*
Make-string returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the string are unspecified.

string *char ...* *R⁵RS Procedure*
Returns a newly allocated string composed of the arguments.

string-length *string* *R⁵RS Procedure*
Returns the number of characters in the given *string*.

string-ref *string k* *R⁵RS Procedure*
String-ref returns character *k* of *string* using zero-origin indexing (*k* must be a valid index of *string*).

string-set! *string k char* *R⁵RS Procedure*
String-set! stores *char* in element *k* of *string* and returns *void* (*k* must be a valid index of *string*).

```
(define (f) (make-string 3 #\*))
(define (g) "****")
(string-set! (f) 0 #\?) ⇒ void
(string-set! (g) 0 #\?) ⇒ error
(string-set! (symbol->string 'immutable)
0
#\?) ⇒ error
```

string=? *string1 string2* *R*⁵*RS* Procedure
string-ci=? *string1 string2* *R*⁵*RS* Procedure

Returns **#t** if the two strings are the same length and contain the same characters in the same positions, otherwise returns **#f**. **String-ci=?** treats upper and lower case letters as though they were the same character, but **string=?** treats upper and lower case as distinct characters.

string<? *string1 string2* *R*⁵*RS* Procedure
string>? *string1 string2* *R*⁵*RS* Procedure
string<=? *string1 string2* *R*⁵*RS* Procedure
string>=? *string1 string2* *R*⁵*RS* Procedure
string-ci<? *string1 string2* *R*⁵*RS* Procedure
string-ci>? *string1 string2* *R*⁵*RS* Procedure
string-ci<=? *string1 string2* *R*⁵*RS* Procedure
string-ci>=? *string1 string2* *R*⁵*RS* Procedure

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

substring *string start end* *R*⁵*RS* Procedure
String must be a string, and **start** and **end** must be exact integers satisfying

$$0 \leq \text{start} \leq \text{end} \leq (\text{string-length string}).$$

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

string-append *string ...* *R*⁵*RS* Procedure
Returns a newly allocated string whose characters form the concatenation of the given strings.

string->list *string* *R*⁵*RS* Procedure
list->string *list* *R*⁵*RS* Procedure
String->list returns a newly allocated list of the characters that make up the given string. **List->string** returns a newly allocated string formed from the characters in the list **list**, which must be a list of characters. **String->list** and **list->string** are inverses so far as **equal?** is concerned.

string-copy *string* *R*⁵*RS* Procedure
Returns a newly allocated copy of the given **string**.

string-split *str* STKLOS Procedure
string-split *str delimiters* STKLOS Procedure

parses **string** and returns a list of tokens ended by a character of the **delimiters** string. If **delimiters** is omitted, it defaults to a string containing a space, a tabulation and a newline characters.

(**string-split** "/usr/local/bin" "/") ⇒ ("usr" "local" "bin")
(**string-split** "once upon a time") ⇒ ("once" "upon" "a" "time")

string-index *str1 str2* STKLOS Procedure
Returns the (first) index where **str1** is a substring of **str2** if it exists; otherwise returns **#f**.

```
(string-index "ca" "abracadabra") ⇒ 4
(string-index "ba" "abracadabra") ⇒ #f
```

string-find? *str1 str2* STKLOS Procedure
Returns **#t** if **str1** appears somewhere in **str2**; otherwise returns **#f**.

string-fill! *string char* STKLOS Procedure
Stores **char** in every element of the given **string** and returns *void*.

The following string primitives are compatible with SRFI-13 (documentation comes from the SRFI-13 document).

string-downcase *str* STKLOS Procedure

string-downcase *str start* STKLOS Procedure

string-downcase *str start end* STKLOS Procedure

Returns a string in which the upper case letters of string **str** between the **start** and **end** indices have been replaced by their lower case equivalent. If **start** is omitted, it defaults to 0. If **end** is omitted, it defaults to the length of **str**.

```
(string-downcase "Foo BAR")      ⇒ "foo bar"
(string-downcase "Foo BAR" 4)    ⇒ "bar"
(string-downcase "Foo BAR" 4 6)  ⇒ "ba"
```

string-downcase! *str* STKLOS Procedure

string-downcase! *str start* STKLOS Procedure

string-downcase! *str start end* STKLOS Procedure

This is the in-place side-effecting variant of **string-downcase**.

```
(string-downcase! (string-copy "Foo BAR") 4) ⇒ "Foo bar"
(string-downcase! (string-copy "Foo BAR") 4 6) ⇒ "Foo baR"
```

string-upcase *str* STKLOS Procedure

string-upcase *str start* STKLOS Procedure

string-upcase *str start end* STKLOS Procedure

Returns a string in which the lower case letters of string **str** between the **start** and **end** indices have been replaced by their upper case equivalent. If **start** is omitted, it defaults to 0. If **end** is omitted, it defaults to the length of **str**.

string-upcase! *str* STKLOS Procedure

string-upcase! *str start* STKLOS Procedure

string-upcase! *str start end* STKLOS Procedure

This is the in-place side-effecting variant of **string-upcase**.

string-titlecase *str* STKLOS Procedure

string-titlecase *str start* STKLOS Procedure

string-titlecase *str start end* STKLOS Procedure

This function returns a string. For every character **c** in the selected range of **str**, if **c** is preceded by a cased character, it is downcased; otherwise it is titlecased. If **start** is omitted, it defaults to 0. If **end** is omitted, it defaults to the length of **str**. Note that if a **start** index is specified, then the character preceding **s[start]** has no effect on the titlecase decision for character **s[start]**.

```
(string-titlecase "--capitalize tHIS sentence.")
  ⇒ "--Capitalize This Sentence."
(string-titlecase "see Spot run. see Nix run.")
  ⇒ "See Spot Run. See Nix Run."
(string-titlecase "3com makes routers.")
  ⇒ "3Com Makes Routers."
(string-titlecase "greasy fried chicken" 2)
  ⇒ "Easy Fried Chicken"
```

string-titlecase! *str*

STKLOS Procedure

string-titlecase! *str start*

STKLOS Procedure

string-titlecase! *str start end*

STKLOS Procedure

This is the in-place side-effecting variant of `string-titlecase`.

string-mutable? *obj*

STKLOS Procedure

Returns `#t` if *obj* is a mutable string, otherwise returns `#f`.

```
(string-mutable? "abc")           ⇒ #f
(string-mutable? (string-copy "abc")) ⇒ #t
(string-mutable? (string #\a #\b #\c)) ⇒ #t
(string-mutable? 12)              ⇒ #f
```

6.8 Vectors

Vectors are heterogenous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time required 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 indexes of a vector are the exact non-negative integers 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.

Vectors are written using the notation `#(obj ...)`. For example, a vector of length 3 containing the number zero in element 0, the list `(2 2 2)` in element 1, and the string "Anna" in element 2 can be written as following:

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

Note: In STKLOS, vectors constants don't need to be quoted.

vector? *obj**R⁵RS* Procedure

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

make-vector *k**R⁵RS* Procedure**make-vector** *k fill**R⁵RS* Procedure

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

vector *obj ...**R⁵RS* Procedure

Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)           ⇒ #(a b c)
```

vector-length *vector**R⁵RS* Procedure

Returns the number of elements in *vector* as an exact integer.

vector-ref *vector k* *R⁵RS* Procedure
k must be a valid index of *vector*. **Vector-ref** returns the contents of element *k* of *vector*.

```
(vector-ref '(1 1 2 3 5 8 13 21)
            5)           ⇒ 8
(vector-ref '(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1))))))
              (if (inexact? i)
                  (inexact->exact i)
                  i))) ⇒ 13
```

vector-set! *vector k obj* *R⁵RS* Procedure
k must be a valid index of *vector*. **Vector-set!** stores *obj* in element *k* of *vector*. The value returned by **vector-set!** is *void*.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue")))
vec)           ⇒ #(0 ("Sue" "Sue") "Anna")

(vector-set! '(0 1 2) 1 "doe") ⇒ error ; constant vector
```

vector->list *vector* *R⁵RS* Procedure

list->vector *list* *R⁵RS* Procedure

Vector->list returns a newly allocated list of the objects contained in the elements of *vector*. **List->vector** returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '(dah dah didah)) ⇒ (dah dah didah)
(list->vector '(dididit dah))   ⇒ #(dididit dah)
```

vector-fill! *vector fill* *R⁵RS* Procedure

Stores *fill* in every element of *vector*. The value returned by **vector-fill!** is *void*.

vector-copy *v* STKLOS Procedure

Return a copy of vector *v*. Note that, if *v* is a constant vector, its copy is not constant.

vector-resize *v size* STKLOS Procedure

vector-resize *v size fill* STKLOS Procedure

Returns a copy of *v* of the given *size*. If *size* is greater than the vector size of *v*, the contents of the newly allocated vector cells is set to the value of *fill*. If *fill* is omitted the content of the new cells is *void*.

vector-mutable? *obj* STKLOS Procedure

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

```
(vector-mutable? '(1 2 a b))           ⇒ #f
(vector-mutable? (vector-copy '(1 2))) ⇒ #t
(vector-mutable? (vector 1 2 3))      ⇒ #t
(vector-mutable? 12)                   ⇒ #f
```

sort *obj predicate* STKLOS Procedure

Obj must be a list or a vector. **Sort** returns a copy of *obj* sorted according to **predicate**. **Predicate** must be a procedure which takes two arguments and returns a true value if the first argument is strictly “before” the second.

```
(sort '(1 2 -4 12 9 -1 2 3) <)
      ⇒ (-4 -1 1 2 2 3 9 12)
(sort #'("one" "two" "three" "four")
      (lambda (x y) (> (string-length x) (string-length y))))
      ⇒ #'("three" "four" "one" "two")
```

6.9 Control features

procedure? *obj*

R⁵RS Procedure

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

```
(procedure? car)           ⇒ #t
(procedure? 'car)         ⇒ #f
(procedure? (lambda (x) (* x x))) ⇒ #t
(procedure? '(lambda (x) (* x x))) ⇒ #f
(call-with-current-continuation procedure?) ⇒ #t
```

apply *proc arg1 ... args*

R⁵RS Procedure

Proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list

```
(append (list arg1 ...) args)
as the actual arguments.
(apply + (list 3 4))           ⇒ 7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)      ⇒ 30
```

map *proc list1 list2 ...*

R⁵RS Procedure

The *lists* must be lists, and *proc* must be a procedure taking as many arguments as there are lists and returning a single value. If more than one list is given, then they must all be the same length. *Map* applies *proc* element-wise to the elements of the *lists* and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the lists is unspecified.

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

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

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

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

for-each *proc list1 list2 ...* *R⁵RS* Procedure

The arguments to **for-each** are like the arguments to **map**, but **for-each** calls *proc* for its side effects rather than for its values. Unlike **map**, **for-each** is guaranteed to call *proc* on the elements of the lists in order from the first element(s) to the last, and the value returned by **for-each** is *void*.

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

every *pred list1 list2 ...* STKLOS Procedure

every applies the predicate *pred* across the lists, returning true if the predicate returns true on every application.

If there are *n* list arguments *list1 ... listn*, then *pred* must be a procedure taking *n* arguments and returning a boolean result.

every applies *pred* to the first elements of the *listi* parameters. If this application returns false, **every** immediately returns **#f**. Otherwise, it iterates, applying *pred* to the second elements of the *listi* parameters, then the third, and so forth. The iteration stops when a false value is produced or one of the lists runs out of values. In the latter case, **every** returns the true value produced by its final application of *pred*. The application of *pred* to the last element of the lists is a tail call.

If one of the *listi* has no elements, **every** simply returns **#t**.

Like **any**, **every**'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (**#t** or **#f**), but a general value.

any *pred list1 list2 ...* STKLOS Procedure

any applies the predicate across the lists, returning true if the predicate returns true on any application.

If there are *n* list arguments *list1 ... listn*, then *pred* must be a procedure taking *n* arguments.

any applies *pred* to the first elements of the *listi* parameters. If this application returns a true value, **any** immediately returns that value. Otherwise, it iterates, applying *pred* to the second elements of the *listi* parameters, then the third, and so forth. The iteration stops when a true value is produced or one of the lists runs out of values; in the latter case, **any** returns **#f**. The application of *pred* to the last element of the lists is a tail call.

Like **every**, **any**'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (**#t** or **#f**), but a general value.

```
(any integer? '(a 3 b 2.7)) ⇒ #t
(any integer? '(a 3.1 b 2.7)) ⇒ #f
(any < '(3 1 4 1 5)
      '(2 7 1 8 2))          ⇒ #t
```

force *promise* *R⁵RS* Procedure

Forces the value of **promise** (see see [\[delay\]](#), page 10). 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.

```

(force (delay (+ 1 2)))           ⇒ 3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p))) ⇒ (3 3)

(define a-stream
  (letrec ((next (lambda (n)
                  (cons n (delay (next (+ n 1))))))
          (next 0)))
  (define head car)
  (define tail (lambda (stream) (force (cdr stream))))

  (head (tail (tail a-stream))) ⇒ 2

```

Force and delay are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

```

(define count 0)
(define p (delay (begin (set! count (+ count 1))
                       (if (> count x)
                           count
                           (force p)))))

(define x 5)
p ⇒ a promise
(force p) ⇒ 6
p ⇒ a promise, still
(begin (set! x 10)
  (force p)) ⇒ 6

```

Note: See *R⁵RS* for details on a possible way to implement force and delay.

call-with-current-continuation *proc*

R⁵RS Procedure

call/cc *proc*

R⁵RS Procedure

Current version of call-with-current-continuation is not conform to *R⁵RS*. Furthermore, the current implementation can lead to a fatal error in some circumstances.

// MUST BE CHANGED FOR NEXT RELEASE

Note: call/cc is just another name for call-with-current-continuation.

values *obj ...*

R⁵RS Procedure

Delivers all of its arguments to its continuation. **Note:** *R⁵RS* imposes to use multiple values in the context of of a call-with-values. In STKLOS, if values is not used with call-with-values, only the first value is used (i.e. others values are *ignored*).

call-with-values *producer consumer*

R⁵RS Procedure

Calls its producer argument with no values and a continuation that, when passed some values, calls the consumer procedure with those values as arguments. The continuation for the call to consumer is the continuation of the call to call-with-values.

```

(call-with-values (lambda () (values 4 5))
  (lambda (a b) b)) ⇒ 5

(call-with-values * -) ⇒ -1

```

receive *<formals>* *<expression>* *<body>* STKLOS Syntax

This form is defined in SRFI-8 ("Binding to Multiple values"). It simplifies the usage of multiple values. Specifically, *<formals>* can have any of three forms:

- (*<variable1>* ... *<variablen>*): The environment in which the receive-expression is evaluated is extended by binding *<variable1>*, ..., *<variablen>* to fresh locations. The *<expression>* is evaluated, and its values are stored into those locations. (It is an error if *<expression>* does not have exactly *n* values.)
- *<variable>*: The environment in which the receive-expression is evaluated is extended by binding *<variable>* to a fresh location. The *<expression>* is evaluated, its values are converted into a newly allocated list, and the list is stored in the location bound to *<variable>*.
- (*<variable1>* ... *<variablen>* . *<variablen + 1>*): The environment in which the receive-expression is evaluated is extended by binding *<variable1>*, ..., *<variablen + 1>* to fresh locations. The *<expression>* is evaluated. Its first *n* values are stored into the locations bound to *<variable1>* ... *<variablen>*. Any remaining values are converted into a newly allocated list, which is stored into the location bound to *<variablen + 1>*. (It is an error if *<expression>* does not have at least *n* values.)

In any case, the expressions in *<body>* are evaluated sequentially in the extended environment. The results of the last expression in the body are the values of the receive-expression.

```
(let ((n 123))
  (receive (q r)
    (values (quotient n 10) (modulo n 10))
    (cons q r)))
⇒ (12 . 3)
```

dynamic-wind *before* *thunk* *after* *R⁵RS* Procedure

Current version of **dynamic-wind** mimics the *R⁵RS* one. In particular, it does not yet interact with **call-with-current-continuation** as required by *R⁵RS*.

Calls **thunk** without arguments, returning the result(s) of this call. **Before** and **after** are called, also without arguments, as required by the following rules (note that in the absence of calls to continuations captured using **call-with-current-continuation** the three arguments are called once each, in order). **Before** is called whenever execution enters the dynamic extent of the call to **thunk** and **after** is called whenever it exits that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. In Scheme, because of **call-with-current-continuation**, the dynamic extent of a call may not be a single, connected time period.

See *R⁵RS* for more details ...

// MUST BE CHANGED for NEXT RELEASE

eval *expression* *environment* *R⁵RS* Procedure

eval *expression* *R⁵RS* Procedure

Current form of STKLOS **eval** is not conform to *R⁵RS*.

// MUST BE CORRECTED FOR NEXT RELEASE

DESCRIBE HERE CALL/EC AND WITH-HANDLER

6.10 Input and Output

R⁵RS states that ports represent input and output devices. However, it defines only ports which are attached to files. In STKLOS, ports can also be attached to strings, to a external command input or output, or even be completely virtual (i.e. the behavior of the port is given by the user).

- String ports are similar to file ports, except that characters are read from (or written to) a string rather than a file.
- External command input or output ports are implemented with Unix pipes and are called *pipe ports*. A pipe port is created by specifying the command to execute prefixed with the string "| ". Specification of a pipe port can occur everywhere a file name is needed.

6.10.1 Ports

call-with-input-file *string proc* *R⁵RS* Procedure

call-with-output-file *string proc* *R⁵RS* Procedure

String should be a string naming a file, and *proc* should be a procedure that accepts one argument. For **call-with-input-file**, the file should already exist. These procedures call *proc* with one argument: the port obtained by opening the named file for input or output. If the file cannot be opened, an error is signaled. If *proc* returns, then the port is closed automatically and the value(s) yielded by the *proc* is(are) returned. If *proc* does not return, then the port will not be closed automatically.

Rationale: Because Scheme's escape procedures have unlimited extent, it is possible to escape from the current continuation but later to escape back in. If implementations were permitted to close the port on any escape from the current continuation, then it would be impossible to write portable code using both **call-with-current-continuation** and **call-with-input-file** or **call-with-output-file**.

call-with-input-string *string proc* STKLOS Procedure

behaves as **call-with-input-file** except that the port passed to *proc* is the string port obtained from *port*.

```
(call-with-input-string "123 456"
  (lambda (x)
    (let* ((n1 (read x))
           (n2 (read x)))
      (cons n1 n2)))) ⇒ (123 . 456)
```

call-with-output-string *proc* STKLOS Procedure

Proc should be a procedure of one argument. **Call-with-output-string** calls *proc* with a freshly opened output string port. The result of this procedure is a string containing all the text that has been written on the string port.

```
(call-with-output-string
  (lambda (x) (write 123 x) (display "Hello" x))) ⇒ "123Hello"
fine (call-with-output-string proc) let ((port (open-output-string))) (proc port) (close-port
port) (get-output-string port)))
; ; String functions ;
c ext read-from-string (read-from-string str)
```

Performs a read from the given *str*. If *str* is the empty string, an end of file object is returned.

```
(read-from-string "123 456") ⇒ 123
(read-from-string "") ⇒ an eof object
```

input-port? *obj* *R⁵RS* Procedure

output-port? *obj* *R⁵RS* Procedure

Returns *#t* if *obj* is an input port or output port respectively, otherwise returns *#f*.

input-string-port? *obj* STKLOS Procedure
output-string-port? *obj* STKLOS Procedure

Returns **#t** if *obj* is an input string port or output string port respectively, otherwise returns **#f**.

input-file-port? *obj* STKLOS Procedure
output-file-port? *obj* STKLOS Procedure

Returns **#t** if *obj* is a file input port or a file output port respectively, otherwise returns **#f**.

interactive-port? *port* STKLOS Procedure

Returns **#t** if *port* is connected to a terminal and **#f** otherwise.

current-input-port *obj* *R⁵RS* Procedure
current-output-port *obj* *R⁵RS* Procedure

Returns the current default input or output port.

current-error-port *obj* STKLOS Procedure

Returns the current default error port.

with-input-from-file *string thunk* *R⁵RS* Procedure
with-output-to-file *string thunk* *R⁵RS* Procedure

String should be a string naming a file, and *proc* should be a procedure of no arguments. For **with-input-from-file**, the file should already exist. The file is opened for input or output, an input or output port connected to it is made the default value returned by **current-input-port** or **current-output-port** (and is used by **(read)**, **(write obj)**, and so forth), and the *thunk* is called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. **With-input-from-file** and **with-output-to-file** return(s) the value(s) yielded by *thunk*.

The following example uses a pipe port opened for reading. It permits to read all the lines produced by an external *ls* command (i.e. the output of the *ls* command is *redirected* to the Scheme pipe port).

```
(with-input-from-file "| ls -ls"
  (lambda ()
    (do ((l (read-line) (read-line)))
        ((eof-object? l))
      (display l)
      (newline))))
```

Hereafter is another example of Unix command redirection. This time, it is the standard input of the Unix command which is redirected.

```
(with-output-to-file "| mail root"
  (lambda ()
    (display "A simple mail from STklos")
    (newline)))
```

with-error-to-file *string thunk* STKLOS Procedure

This procedure is similar to **with-output-to-file**, excepted that it uses the current error port instead of the output port.

with-input-from-string *string thunk* STKLOS Procedure

A string port is opened for input from *string*. `Current-input-port` is set to the port and *thunk* is called. When *thunk* returns, the previous default input port is restored. `With-input-from-string` returns the value(s) computed by *thunk*.

```
(with-input-from-string "123 456"
 (lambda () (read)))           ⇒ 123
```

with-output-to-string *thunk* STKLOS Procedure

A string port is opened for output. `Current-output-port` is set to it and *thunk* is called. When *thunk* returns, the previous default output port is restored. `With-output-to-string` returns the string containing the text written on the string port.

```
(with-output-to-string
 (lambda () (write 123) (write "Hello"))) ⇒ "123\Hello\""
```

open-input-file *filename* *R⁵RS* Procedure

Takes a string naming an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened, an error is signalled.

Note: if *filename* starts with the string "| ", this procedure returns a pipe port. Consequently, it is not possible to open a file whose name starts with those two characters.

open-input-string *str* STKLOS Procedure

Returns an input string port capable of delivering characters from *str*.

open-output-file *filename* *R⁵RS* Procedure

Takes a string naming an output file to be created and returns an output port capable of writing characters to a new file by that name. If the file cannot be opened, an error is signalled. If a file with the given name already exists, it is rewritten.

Note: if *filename* starts with the string "| ", this procedure returns a pipe port. Consequently, it is not possible to open a file whose name starts with those two characters.

open-output-string STKLOS Procedure

Returns an output string port capable of receiving and collecting characters.

open-file *filename mode* STKLOS Procedure

Opens the file whose name is *filename* with the specified string *mode* which can be:

- "r" to open file for reading. The stream is positioned at the beginning of the file.
- "r+" to open file for reading and writing. The stream is positioned at the beginning of the file.
- "w" to truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
- "w+" to open file for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- "a" to open for writing. The file is created if it does not exist. The stream is positioned at the end of the file.
- "a+" to open file for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file.

If the file can be opened, `open-file` returns the port associated with the given file, otherwise it returns `#f`. Here again, the "magic" string "| " permits to open a pipe port (in this case mode can only be "r" or "w").

get-output-string *port* STKLOS Procedure
 Returns a string containing all the text that has been written on the output string *port*.

```
(let ((p (open-output-string)))
  (display "Hello, world" p)
  (get-output-string p))      ⇒ "Hello, world"
```

close-input-port *port* *R⁵RS* Procedure
close-output-port *port* *R⁵RS* Procedure

Closes the port associated with *port*, rendering the port incapable of delivering or accepting characters. These routines have no effect if the port has already been closed. The value returned is *void*.

close-port *port* STKLOS Procedure
 Closes the port associated with *port*.

rewind-file-port *port* STKLOS Procedure
 Sets the port position to the beginning of *port*. The value returned by **rewind-port** is *void*.

port-current-line *R⁵RS* Procedure
port-current-line *port* *R⁵RS* Procedure

Returns the current line number associated to the given input *port* as an integer. The *port* argument may be omitted, in which case it defaults to the value returned by **current-input-port**.

port-file-name *port* STKLOS Procedure
 Returns the file name used to open *port*; *port* must be a file port.

port-idle-register! *port thunk* STKLOS Procedure
port-idle-unregister! *port thunk* STKLOS Procedure
port-idle-reset! *port* STKLOS Procedure

port-idle-register! allows to register *thunk* as an idle handler when reading on *port*. That means that *thunk* will be called continuously while waiting an input on *port* (and only while using a reading primitive on this port). **port-idle-unregister!** can be used to unregister a handler previously set by **port-idle-register!**. The primitive **port-idle-reset!** unregisters all the handlers set on *port*.

Hereafter is a (not too realistic) example: a message will be displayed repeatedly until a character is read on the current input port.

```
(let ((idle (lambda () (display "Nothing to read!\n"))))
  (port-idle-register! (current-input-port) idle)
  (let ((result (read)))
    (port-idle-unregister! (current-input-port) idle)
    result))
```

6.10.2 Input

read *R⁵RS* Procedure
read *port* *R⁵RS* Procedure

Read converts external representations of Scheme objects into the objects themselves. **Read** returns the next object parsable from the given input *port*, updating *port* to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned. The port remains open, and further attempts to read will also return an end of file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The port argument may be omitted, in which case it defaults to the value returned by `current-input-port`. It is an error to read from a closed port.

read-with-shared-structure *R⁵RS* Procedure

read-with-shared-structure *port* *STKLOS* Procedure

`read-with-shared-structure` is identical to `read`. It has been added to be compatible with SRFI-38. STKLOS always knew how to deal with recursive data.

read-char *R⁵RS* Procedure

read-char *port* *R⁵RS* Procedure

Returns the next character available from the input `port`, updating the `port` to point to the following character. If no more characters are available, an end of file object is returned. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

peek-char *R⁵RS* Procedure

peek-char *port* *R⁵RS* Procedure

Returns the next character available from the input `port`, without updating the port to point to the following character. If no more characters are available, an end of file object is returned. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

Note: The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` with the same port. The only difference is that the very next call to `read-char` or `peek-char` on that port will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

eof-object? *obj* *R⁵RS* Procedure

Returns `#t` if `obj` is an end of file object, otherwise returns `#f`.

char-ready? *R⁵RS* Procedure

char-ready? *port* *R⁵RS* Procedure

Returns `#t` if a character is ready on the input port and returns `#f` otherwise. If `char-ready` returns `#t` then the next `read-char` operation on the given port is guaranteed not to hang. If the port is at end of file then `char-ready?` returns `#t`. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

read-line *R⁵RS* Procedure

read-line *port* *R⁵RS* Procedure

Reads the next line available from the input port `port`. This function returns 2 values: the first one is the string which contains the line read, and the second one is the end of line delimiter. The end of line delimiter can be an end of file object, a character or a string in case of a multiple character delimiter. If no more characters are available on `port`, an end of file object is returned. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

Note: As said in See [Multiple values], page 40, if `read-line` is not used in the context of `call-with-values`, the second value return by this procedure is ignored.

port->string *port* STKLOS Procedure
port->sexp-list *port* STKLOS Procedure
port->string-list *port* STKLOS Procedure

All these procedure take a port opened for reading. **Port->string** reads *port* until the it reads an end of file object and returns all the characters read as a string. **Port->sexp-list** and **port->string-list** do the same things except that they return a list of S-expressions and a list of strings respectively. For the following example we suppose that file "foo" is formed of two lines which contains respectively the number 100 and the string "bar".

```
(port->sexp-list (open-input-file "foo")) ⇒ (100 "bar")
(port->string-list (open-input-file "foo")) ⇒ ("100" "\"bar\"")
```

6.10.3 Output

write *obj* *R⁵RS* Procedure
write *obj port* *R⁵RS* Procedure

Writes a written representation of *obj* to the given *port*. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the `#\` notation. **Write** returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by **current-output-port**.

write* *obj* STKLOS Procedure
write* *obj port* STKLOS Procedure

Writes a written representation of *obj* to the given *port*. The main difference with the **write** procedure is that **write*** handles data structures with cycles. Circular structure written by this procedure use the “`#n=`” and “`#n#`” notations (see [\[circular structure\]](#), page 2).

write-with-shared-structure *obj* STKLOS Procedure
write-with-shared-structure *obj port* STKLOS Procedure
write-with-shared-structure *obj port optarg* STKLOS Procedure

write-with-shared-structure has been added to be compatible with SRFI-38. It is identical to **write***, except that it accepts one more parameter (*optarg*). This parameter, which is not specified in SRFI-38, is always ignored.

display *obj* *R⁵RS* Procedure
display *obj port* *R⁵RS* Procedure

Writes a representation of *obj* to the given *port*. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by **write-char** instead of by **write**. **Display** returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by **current-output-port**.

Rationale: **Write** is intended for producing machine-readable output and **display** is for producing human-readable output.

newline *R⁵RS* Procedure
newline *port* *R⁵RS* Procedure

Writes an end of line to *port*. Exactly how this is done differs from one operating system to another. Returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by **current-output-port**.

write-char *char* *R⁵RS* Procedure
write-char *char port* *R⁵RS* Procedure

Writes the character **char** (not an external representation of the character) to the given **port** and returns an unspecified value. The **port** argument may be omitted, in which case it defaults to the value returned by **current-output-port**.

format *port str obj ...* STKLOS Procedure
format *str obj* STKLOS Procedure

Writes the **objs** to the given **port**, according to the format string **str**. **Str** is written literally, except for the following sequences:

- **~a** or **~A** is replaced by the printed representation of the next **obj**.
- **~s** or **~S** is replaced by the “slashified” printed representation of the next **obj**.
- **~w** or **~W** is replaced by the printed representation of the next **obj** (circular structures are correctly handled and printed using **writes***).
- **~~** is replaced by a single tilde character.
- **~%** is replaced by a newline

Port can be a boolean or a port. If **port** is **#t**, output goes to the current output port; if **port** is **#f**, the output is returned as a string. Otherwise, the output is printed on the specified port.

```
(format #f "A test.")      ⇒ "A test."
(format #f "A ~a." "test") ⇒ "A test."
(format #f "A ~s." "test") ⇒ "A \"test\"."
```

The second form of **format** is compliant with SRFI-28. That is, when **port** is omitted, the output is returned as a string as if **port** was given the value **#f**.

flush STKLOS Procedure
flush *port* STKLOS Procedure

Flushes the buffer associated with the given output **port**. The **port** argument may be omitted, in which case it defaults to the value returned by **current-output-port**

6.10.4 System interface

load *filename* *R⁵RS* Procedure

Filename should be a string naming an existing file containing Scheme expressions. **Load** has been extended in STKLOS to allow loading of file containing Scheme compiled code as well as object files (*aka* shared objects). The loading of object files is not available on all architectures. The value returned by **load** is *void*.

If the file whose name is **filename** cannot be located, **load** will try to find it in one of the directories given by **(get-load-path)** with the suffixes given by **(get-load-suffixes)**.

try-load *filename* STKLOS Procedure

try-load tries to load the file named **filename**. As **load**, **try-load** tries to find the file given the current load path and a set of suffixes if **filename** cannot be loaded. If **try-load** is able to find a readable file, it is loaded, and **try-load** returns **#t**. Otherwise, **try-load** returns **#f**.

load-path STKLOS Procedure

Returns the current load path. The load path is a list of strings which correspond to the directories in which a file must be searched for loading. Directories of the load path are

prepended (in their apparition order) to the file name given to `load` or `try-load` until the file can be loaded.

The initial value of the current load path can be set from the shell, by setting the `STKLOS_LOAD_PATH` variable.

set-load-path! *new-path* STKLOS Procedure
Sets the current load path to the list of strings given in *new-path*.

load-suffixes STKLOS Procedure
Returns the list of possible suffixes for a Scheme file. Each suffix, must be a string. Suffixes are appended (in their apparition order) to a file name is appended to a file name given to `load` or `try-load` until the file can be loaded.

set-load-suffixes! *suffixes* STKLOS Procedure
Sets the possible suffixes to the list of strings given in *suffixes*.

find-path *str* STKLOS Procedure

find-path *str path* STKLOS Procedure

find-path *str path suffixes* STKLOS Procedure

In its first form, `find-path` returns the path name of the file that should be loaded by the procedure `load` given the name *str*. The string returned depends of the current load path and of the currently accepted suffixes. The other forms of `find-path` are more general and allow to give a path list (a list of strings representing supposed directories) and a set of suffixes (given as a list of strings too) to try for finding a file. If no file is found, `find-path` returns `#f`.

For instance, on a "classical" Unix box:

```
(find-path "passwd" '("/bin" "/etc" "/tmp"))
⇒ "/etc/passwd"
(find-path "stdio" '("/usr" "/usr/include") '("c" "h" "stk"))
⇒ "/usr/include/stdio.h"
```

require *string* STKLOS Procedure

provide *string* STKLOS Procedure

provided? *string* STKLOS Procedure

`require` loads the file whose name is *string* if it was not previously “*provided*”. `provide` permits to store *string* in the list of already provided files. Providing a file permits to avoid subsequent loads of this file. `provided?` returns `#t` if *string* was already provided; it returns `#f` otherwise.

6.11 Keywords

Keywords are symbolic constants which evaluate to themselves. A keyword is a symbol whose first (or last) character is a colon (":").

keyword *obj* STKLOS Procedure

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

```
(keyword? 'foo)      ⇒ #f
(keyword? ':foo)    ⇒ #t
(keyword? 'foo:)    ⇒ #t
(keyword? :foo)     ⇒ #t
(keyword? foo:)     ⇒ #t
```

make-keyword *s* STKLOS Procedure

Builds a keyword from the given *s*. The parameter *s* must be a symbol or a string.

```
(make-keyword "test")    ⇒ :test
(make-keyword 'test)    ⇒ :test
(make-keyword ":hello") ⇒ ::hello
```

keyword->string *key* STKLOS Procedure

Returns the name of *key* as a string. The result does not contain a colon.

key-get *list key* STKLOS Procedure

key-get *list key default* STKLOS Procedure

List must be a list of keywords and their respective values. **key-get** scans the *list* and returns the value associated with the given *key*. If *key* does not appear in an odd position in *list*, the specified *default* is returned, or an error is raised if no *default* was specified.

```
(key-get '(:one 1 :two 2) :one)    ⇒ 1
(key-get '(:one 1 :two 2) :four #f) ⇒ #f
(key-get '(:one 1 :two 2) :four)   ⇒ error
```

key-set! *list key value* STKLOS Procedure

List must be a list of keywords and their respective values. **key-set!** sets the value associated to *key* in the keyword list. If the key is already present in *list*, the keyword list is *physically* changed.

```
(let ((l '(:one 1 :two 2)))
  (set! l (key-set! l :three 3))
  (cons (key-get l :one)
        (key-get l :three)))    ⇒ (1 . 3)
```

key-delete! *list key* STKLOS Procedure

List must be a list of keywords and their respective values. **key-delete** remove the *key* and its associated value of the keyword list. The key can be absent of the list.

key-delete! does the same job than **key-delete** by physically modifying its *list* argument.

```
(key-delete '(:one 1 :two 2) :one) ⇒ (:one 1)
(key-delete '(:one 1 :two 2) :three) ⇒ (:one 1 :two 2)
```

6.12 Regular Expressions

STKLOS uses the Philip Hazel's Perl-compatible Regular Expression (PCRE) library for implementing regexps [PCRE01]. Consequently, the STKLOS regular expression syntax is the same as PCRE, and Perl by the way.

The following text is extracted from the PCRE package. However, to make things shorter, some of the original documentation as not been reported here. In particular some possibilities of PCRE have been completely occulted (those whose description was too long and which seems (at least to me), not too important). Read the documentation provided with PCRE for a complete description⁴.

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

⁴ The latest release of PCRE is available from <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre>

The quick brown fox

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *meta-characters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

<code>\</code>	general escape character with several uses
<code>^</code>	assert start of subject (or line, in multiline mode)
<code>\$</code>	assert end of subject (or line, in multiline mode)
<code>.</code>	match any character except newline (by default)
<code>[</code>	start character class definition
<code> </code>	start of alternative branch
<code>(</code>	start subpattern
<code>)</code>	end subpattern
<code>?</code>	extends the meaning of (also 0 or 1 quantifier also quantifier minimizer
<code>*</code>	0 or more quantifier
<code>+</code>	1 or more quantifier
<code>{</code>	start min/max quantifier

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-characters are:

<code>\</code>	general escape character
<code>^</code>	negate the class, but only if the first character
<code>-</code>	indicates character range
<code>]</code>	terminates the character class

The following sections describe the use of each of the meta-characters.

Backslash

The backslash character has several uses. Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

Another use of backslash is for specifying generic character types:

<code>\d</code>	any decimal digit
<code>\D</code>	any character that is not a decimal digit
<code>\s</code>	any whitespace character
<code>\S</code>	any character that is not a whitespace character
<code>\w</code>	any "word" character
<code>\W</code>	any "non-word" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a "word".

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

Circumflex and Dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see below).

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar has no special meaning in a character class.

The meanings of the circumflex and dollar characters are changed if the MULTILINE option is set. When this is the case, they match immediately after and immediately before an internal "\n" character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern `^abc$` matches the subject string `"def\nabc"` in multiline mode, but not otherwise.

Dot

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the DOTALL option is set, dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

Square Braquets

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lower case vowel, while `[^aeiou]` matches any character that is not a lower case vowel.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches `"A"` as well as `"a"`, and a caseless `[^aeiou]` does not match `"A"`, whereas a careful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the DOTALL or MULTILINE options is. A class such as `[^a]` will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between `d` and `m`, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to `[^_ 'wxyzabc]`, matched caselessly.

The character types `\d`, `\D`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[\^W_]` matches any letter or digit, but not underscore.

POSIX Character Classes

PCRE supports the POSIX notation for character classes, which uses names enclosed by `[:` and `:]` within the enclosing square brackets.

```
[01[:alpha:]]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are

<code>alnum</code>	letters and digits
<code>alpha</code>	letters
<code>ascii</code>	character codes 0 - 127
<code>cntrl</code>	control characters
<code>digit</code>	decimal digits (same as <code>\d</code>)
<code>graph</code>	printing characters, excluding space
<code>lower</code>	lower case letters
<code>print</code>	printing characters, including space
<code>punct</code>	printing characters, excluding letters and digits
<code>space</code>	white space (same as <code>\s</code>)
<code>upper</code>	upper case letters
<code>word</code>	"word" characters (same as <code>\w</code>)
<code>xdigit</code>	hexadecimal digits

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern affects only that part of the subpattern that follows it, so

```
(a(?i)b)c
```

matches `abc` and `aBc` and no other strings (assuming `CASELESS` is not used).

Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cactaract", or "caterpillar". Without the parentheses, it would match "cactaract", "erpillar" or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above). Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3.

Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a single character, possibly escaped

- the `.` metacharacter
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion - see below)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

```
* is      equivalent to {0,}
+ is      equivalent to {1,}
? is      equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences `/*` and `*/` and within the sequence, individual `*` and `/` characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first command */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the `.*` item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

Back References

Outside a character class, a backslash followed by a digit is a back reference to a capturing subpattern earlier (i.e. to its left) in the pattern, provided there have been that many previous capturing left parentheses.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If careful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `^` and `$` are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with `(?=` for positive assertions and `(?!` for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion `(?!foo)` is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

Lookbehind assertions start with `(?<=` for positive assertions and `(?<!` for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only subpatterns.

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?<=\d{3}...)(?!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999". Assertions count towards the maximum of 200 parenthesized subpatterns.

Other features

As said before, only a subset of PCRE is described in this document. In particular points such as

- Once only subpatterns
- Conditionals subpatterns
- Comments
- Recursive patterns

will not be discussed here. See the PCRE documentation for more details.

string->regex *string* STKLOS Procedure
String->regex takes a string representation of a regular expression and compiles it into a regex value. Other regular expression procedures accept either a string or a regex value as the matching pattern. If a regular expression string is used multiple times, it is faster to compile the string once to a regex value and use it for repeated matches instead of using the string each time.

regex? *obj* STKLOS Procedure
Regex returns **#t** if *obj* is a regex value created by the **regex**, otherwise **regex** returns **#f**.

regex-match *pattern str* STKLOS Procedure
regex-match-positions *pattern str* STKLOS Procedure

These functions attempt to match **pattern** (a string or a regex value) to **str**. If the match fails, **#f** is returned. If the match succeeds, a list (containing strings for **regex-match** and positions for **regex-match-positions**) is returned. The first string (or positions) in this list is the portion of string that matched pattern. If two portions of string can match pattern, then the earliest and longest match is found, by default.

Additional strings or positions are returned in the list if **pattern** contains parenthesized sub-expressions; matches for the sub-expressions are provided in the order of the opening parentheses in **pattern**.

```

(regexp-match-positions "ca" "abracadabra")
  ⇒ ((4 6))
(regexp-match-positions "CA" "abracadabra")
  ⇒ #f
(regexp-match-positions "(?i)CA" "abracadabra")
  ⇒ ((4 6))
(regexp-match "(a*)(b*)(c*)" "abc")
  ⇒ ("abc" "a" "b" "c")
(regexp-match-positions "(a*)(b*)(c*)" "abc")
  ⇒ ((0 3) (0 1) (1 2) (2 3))
(regexp-match-positions "(a*)(b*)(c*)" "c")
  ⇒ ((0 1) (0 0) (0 0) (0 1))
(regexp-match "(?<=\\d{3})(?!999)foo" "999foo and 123foo")
  ⇒ ((14 17))

```

regexp-replace *pattern string substitution*

STKLOS Procedure

regexp-replace-all *pattern string substitution*

STKLOS Procedure

Regexp-replace matches the regular expression **pattern** against **string**. If there is a match, the portion of **string** which matches **pattern** is replaced by the **substitution** string. If there is no match, **regexp-replace** returns **string** unmodified. Note that the given **pattern** could be here either a string or a regular expression.

If **pattern** contains `\n` where **n** is a digit between 1 and 9, then it is replaced in the substitution with the portion of string that matched the **n**-th parenthesized subexpression of **pattern**. If **n** is equal to 0, then it is replaced in **substitution** with the portion of **string** that matched **pattern**.

Regexp-replace replaces the first occurrence of **pattern** in **string**. To replace **all** the occurrences of **pattern**, use **regexp-replace-all**.

```

(regexp-replace "a*b" "aaabbcccc" "X")
  ⇒ "Xbcccc"
(regexp-replace (string->regexp "a*b") "aaabbcccc" "X")
  ⇒ "Xbcccc"
(regexp-replace "(a*)b" "aaabbcccc" "X\\1Y")
  ⇒ "XaaaYbcccc"
(regexp-replace "f(.*?)r" "foobar" "\\1 \\1")
  ⇒ "ooba ooba"
(regexp-replace "f(.*?)r" "foobar" "\\0 \\0")
  ⇒ "foobar foobar"

(regexp-replace "a*b" "aaabbcccc" "X")
  ⇒ "Xbcccc"
(regexp-replace-all "a*b" "aaabbcccc" "X")
  ⇒ "XXcccc"

```

regexp-quote *str*

STKLOS Procedure

Takes an arbitrary string and returns a string where characters of **str** that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

```

(regexp-quote "cons")      ⇒ "cons"
(regexp-quote "list?")    ⇒ "list\\"

```

regexp-quote is useful when building a composite regexp from a mix of regexp strings and verbatim strings.

6.13 Pattern Matching

Pattern matching is a key feature of most modern functional programming languages since it allows clean and secure code to be written. Internally, “pattern-matching forms” should be translated (compiled) into cascades of “elementary tests” where code is made as efficient as possible, avoiding redundant tests; STKLOS’s “pattern matching compiler” provides this⁵. The technique used is described in details in [QueinnecGeffroy92], and the code generated can be considered optimal⁶ due to the way this “pattern compiler” was obtained.

The “pattern language” allows the expression of a wide variety of patterns, including:

- Non-linear patterns: pattern variables can appear more than once, allowing comparison of subparts of the datum (through `eq?`)
- Recursive patterns on lists: for example, checking that the datum is a list of zero or more `as` followed by zero or more `bs`.
- Pattern matching on lists as well as on vectors.

6.13.1 STKLOS pattern matching facilities

Only two special forms are provided for this in STKLOS: `match-case` and `match-lambda`.

match-case *key clause...*

STKLOS syntax

The argument *key* may be any expression and each *clause* has the form

```
(pattern s-expression...)
```

Semantics: A `match-case` expression is evaluated as follows. *key* is evaluated and the result is compared with each successive pattern. If the pattern in some *clause* yields a match, then the expressions in that *clause* are evaluated from left to right in an environment where the pattern variables are bound to the corresponding subparts of the datum, and the result of the last expression in that *clause* is returned as the result of the `match-case` expression. If no *pattern* in any *clause* matches the datum, then, if there is an `else` clause, its expressions are evaluated and the result of the last is the result of the whole `match-case` expression; otherwise the result of the `match-case` expression is unspecified.

The equality predicate used is `eq?`.

```
(match-case '(a b a)
  ((?x ?x) 'foo)
  ((?x ?- ?x) 'bar))
⇒ bar
```

The following syntax is also available:

match-lambda *clause...*

STKLOS syntax

It expands into a lambda-expression expecting an argument which, once applied to an expression, behaves exactly like a `match-case` expression.

```
((match-lambda
  ((?x ?x) 'foo)
  ((?x ?- ?x) 'bar))
 '(a b a))
⇒ bar
```

⁵ The “pattern matching compiler” has been written by Jean-Marie Geffroy and is part of the Manuel Serrano’s Bigloo compiler [Serrano01] since several years. The code (and documentation) included in STKLOS has been stolen from the Bigloo package v2.4 (the only difference between both package is the pattern matching of structures which is absent in STKLOS).

⁶ In the cases of pattern matching in lists and vectors, not in structures for the moment.

6.13.2 The pattern language

The syntax for <pattern> is:

<pattern> \mapsto	<i>Matches:</i>
<atom>	the <atom>.
(kwote <atom>)	any expression eq? to <atom>.
(and <pat1> ... <patn>)	if all of <pati> match.
(or <pat1><patn>)	if any of <pat1> through <patn> matches.
(not <pat>)	if <pat> doesn't match.
(? <predicate>)	if <predicate> is true.
(<pat1> ... <patn>)	a list of n elements. Here, ... is a meta-character denoting a finite repetition of patterns.
<pat> ...	a (possibly empty) repetition of <pat> in a list.
#(<pat> ... <patn>)	a vector of n elements.
?<id>	anything, and binds id as a variable.
?-	anything.
??-	any (possibly empty) repetition of anything in a list.
???-	any end of list.

Remark: and, or, not, check and kwote must be quoted in order to be treated as literals. This is the only justification for having the kwote pattern since, by convention, any atom which is not a keyword is quoted.

Explanations through examples

- ?- matches any s-expr
- a matches the atom 'a.
- ?a matches any expression, and binds the variable a to this expression.
- (? integer?) matches any integer
- (a (a b)) matches the only list '(a (a b)).
- ???- can only appear at the end of a list, and always succeeds. For instance, (a ???-) is equivalent to (a . ?-).
- when occurring in a list, ??- matches any sequence of anything: (a ??- b) matches any list whose car is a and last car is b.
- (a ...) matches any list of a's, possibly empty.
- (?x ?x) matches any list of length 2 whose car is eq to its cadr
- ((and (not a) ?x) ?x) matches any list of length 2 whose car is not eq to 'a but is eq to its cadr
- #(?- ?- ???-) matches any vector whose length is at least 2.

Remark: ??- and ... patterns can not appear inside a vector, where you should use ???-: For example, #(a ??- b) or #(a ...) are invalid patterns, whereas #(a ???-) is valid and matches any vector whose first element is the atom a.

6.14 Hash Tables

A hash table consists of zero or more entries, each consisting of a key and a value. Given the key for an entry, the hashing function can very quickly locate the entry, and hence the

corresponding value. There may be at most one entry in a hash table with a particular key, but many entries may have the same value.

STKLOS hash tables grow gracefully as the number of entries increases, so that there are always less than three entries per hash bucket, on average. This allows for fast lookups regardless of the number of entries in a table.

make-hash-table STKLOS Procedure
make-hash-table *comparison* STKLOS Procedure
make-hash-table *comparison hash* STKLOS Procedure

Make-hash-table admits three different forms. The most general form admit two arguments. The first argument is a comparison function which determines how keys are compared; the second argument is a function which computes a hash code for an object and returns the hash code as a non negative integer. Objects with the same hash code are stored in an A-list registered in the bucket corresponding to the key.

If omitted,

- **hash** defaults to the **hash-table-hash** procedure (see see [\[hash-table-hash\]](#), page 60).
- **comparison** defaults to the **eq?** procedure (see see [\[eq?\]](#), page 18).

Consequently,

```
(define h (make-hash-table))
```

is equivalent to

```
(define h (make-hash-table eq? hash-table-hash))
```

An interesting example is

```
(define h (make-hash-table string-ci=? string-length))
```

which defines a new hash table which uses **string-ci=?** for comparing keys. Here, we use the **string-length** as a (very simple) hashing function. Of course, a function which gives a key depending of the characters composing the string gives a better repartition and should probably enhance performances. For instance, the following call to **make-hash-table** should return a more efficient, even if not perfect, hash table:

```
(make-hash-table
  string-ci=?
  (lambda (s)
    (let ((len (string-length s)))
      (do ((h 0) (i 0 (+ i 1)))
          ((= i len) h)
        (set! h (+ h (char->integer
                     (char-downcase (string-ref s i))))))))))
```

Note: Hash tables with a comparison function equal to **eq?** or **string=?** are handled in an more efficient way (in fact, they don't use **hash-table-hash** function to speed up hash table retrievals).

hash-table? *obj* STKLOS Procedure
 Returns **#t** if *obj* is a hash table, returns **#f** otherwise.

hash-table-hash *obj* STKLOS Procedure
 Computes a hash code for an object and returns this hash code as a non negative integer. A property of **hash-table-hash** is that

```
(equal? x y) ⇒ (equal? (hash-table-hash x) (hash-table-hash y))
```

as the the Common Lisp **sxhash** function from which this procedure is modeled.

hash-table-put! *hash key value* STKLOS Procedure

Enters an association between *key* and *value* in the *hash* table. The value returned by *hash-table-put!* is *void*.

hash-table-get *hash key* STKLOS Procedure

hash-table-get *hash key default* STKLOS Procedure

Returns the value associated with *key* in the given *hash* table. If no value has been associated with *key* in *hash*, the specified *default* is returned if given; otherwise an error is raised.

```
(define h1 (make-hash-table))
(hash-table-put! h1 'foo (list 1 2 3))
(hash-table-get h1 'foo)           ⇒ (1 2 3)
(hash-table-get h1 'bar 'absent)   ⇒ absent
(hash-table-get h1 'bar)           ⇒ error
(hash-table-put! h1 '(a b c) 'present)
(hash-table-get h1 '(a b c) 'absent) ⇒ absent

(define h2 (make-hash-table equal?))
(hash-table-put! h2 '(a b c) 'present)
(hash-table-get h2 '(a b c))       ⇒ present
```

hash-table-remove *hash key* STKLOS Procedure

Deletes the entry for *key* in *hash*, if it exists. Result of *hash-table-remove!* is *void*.

```
(define h (make-hash-table))
(hash-table-put! h 'foo (list 1 2 3))
(hash-table-get h 'foo)           ⇒ (1 2 3)
(hash-table-remove! h 'foo)
(hash-table-get h 'foo 'absent)   ⇒ absent
```

hash-table-update! *hash key update-fun init-value* STKLOS Procedure

Update the value associated to *key* in table *hash* if *key* is already in table with the value (*update-fun current-value*). If no value is associated to *key*, a new entry in the table is inserted with the *init-value* associated to it.

```
(let ((h (make-hash-table))
      (1+ (lambda (n) (+ n 1))))
  (hash-table-update! h 'test 1+ 100)
  (hash-table-update! h 'test 1+ 100)
  (hash-table-get h 'test))       ⇒ 101
```

hash-table-for-each *hash proc* STKLOS Procedure

Proc must be a procedure taking two arguments. *Hash-table-for-each* calls *proc* on each key/value association in *hash*, with the key as the first argument and the value as the second. The value returned by *hash-table-for-each* is *void*.

Note: The order of application of *proc* is unspecified.

```
(let ((h (make-hash-table))
      (sum 0))
  (hash-table-put! h 'foo 2)
  (hash-table-put! h 'bar 3)
  (hash-table-for-each h (lambda (key value)
                          (set! sum (+ sum value))))
  sum)                             ⇒ 5
```

hash-table-map *hash proc* STKLOS Procedure

Proc must be a procedure taking two arguments. **Hash-table-map** calls **proc** on each key/value association in **hash**, with the key as the first argument and the value as the second. The result of **hash-table-map** is a list of the values returned by **proc**, in an unspecified order.

Note: The order of application of **proc** is unspecified.

```
(let ((h (make-hash-table)))
  (dotimes (i 5)
    (hash-table-put! h i (number->string i)))
  (hash-table-map h (lambda (key value)
                     (cons key value))))
⇒ ((3 . "3") (4 . "4") (0 . "0") (1 . "1") (2 . "2"))
```

hash-table->list *hash* STKLOS Procedure

Returns an “association list” built from the entries in **hash**. Each entry in **hash** will be represented as a pair whose **car** is the entry’s key and whose **cdr** is its value.

Note: the order of pairs in the resulting list is unspecified.

```
(let ((h (make-hash-table)))
  (dotimes (i 5)
    (hash-table-put! h i (number->string i)))
  (hash-table->list h))
⇒ ((3 . "3") (4 . "4") (0 . "0") (1 . "1") (2 . "2"))
```

hash-table-stats *hash* STKLOS Procedure

hash-table-stats *hash port* STKLOS Procedure

Prints overall information about **hash**, such as the number of entries it contains, the number of buckets in its hash array, and the utilization of the buckets. Informations are printed on **port**. If no **port** is given to **hash-table-stats**, information are printed on the current output port (see see [\[current-output-port\]](#), page 43).

6.15 Processes

STKLOS provides access to Unix processes as first class objects. Basically, a process contains several informations such as the standard system process identification (aka PID on Unix Systems), the files where the standard files of the process are redirected.

run-process *command p1 p2 ...* STKLOS Procedure

run-process creates a new process and run the executable specified in **command**. The **p** correspond to the command line arguments. The following values of **p** have a special meaning:

- **:input** permits to redirect the standard input file of the process. Redirection can come from a file or from a pipe. To redirect the standard input from a file, the name of this file must be specified after **:input**. Use the special keyword **:pipe** to redirect the standard input from a pipe.
- **:output** permits to redirect the standard output file of the process. Redirection can go to a file or to a pipe. To redirect the standard output to a file, the name of this file must be specified after **:output**. Use the special keyword **:pipe** to redirect the standard output to a pipe.
- **:error** permits to redirect the standard error file of the process. Redirection can go to a file or to a pipe. To redirect the standard error to a file, the name of this file must be specified after **error**. Use the special keyword **:pipe** to redirect the standard error to a pipe.

- `:wait` must be followed by a boolean value. This value specifies if the process must be run asynchronously or not. By default, the process is run asynchronously (i.e. `:wait` is `#f`).
- `:host` must be followed by a string. This string represents the name of the machine on which the command must be executed. This option uses the external command `rsh`. The shell variable `PATH` must be correctly set for accessing it without specifying its absolute path.
- `:fork` must be followed by a boolean value. This value specifies if a `fork` system call must be done before running the process. If the process is run without `fork` the Scheme program is lost. This feature mimics the “`exec`” primitive of the Unix shells. By default, a fork is executed before running the process (i.e. `:fork` is `#t`). This option works on Unix implementations only.

The following example launches a process which executes the Unix command `ls` with the arguments `-l` and `/bin`. The lines printed by this command are stored in the file `/tmp/X`

```
(run-process "ls" "-l" "/bin" :output "/tmp/X")
```

process? *obj* STKLOS Procedure

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

process-alive? *proc* STKLOS Procedure

Returns `#t` if process *proc* is currently running, otherwise returns `#f`.

process-pid *proc* STKLOS Procedure

Returns an integer which represents the Unix identification (PID) of the process.

process-input *proc* STKLOS Procedure

process-input *proc* STKLOS Procedure

process-input *proc* STKLOS Procedure

Returns the file port associated to the standard input, output or error of *proc*, if it is redirected in (or to) a pipe; otherwise returns `#f`. Note that the returned port is opened for reading when calling `process-output` or `process-error`; it is opened for writing when calling `process-input`.

process-wait *proc* STKLOS Procedure

Stops the current process (the Scheme process) until *proc* completion. `Process-wait` returns `#f` when *proc* is already terminated; it returns `#t` otherwise.

process-exit-status *proc* STKLOS Procedure

Returns the exit status of *proc* if it has finished its execution; returns `#f` otherwise.

process-send-signal *proc sig* STKLOS Procedure

Sends the integer signal *sig* to *proc*. Since value of *sig* is system dependant, use the symbolic defined signal constants to make your program independant of the running system (see see [\[signals\]](#), [page 74](#)). The result of `process-send-signal` is *void*.

process-kill *proc* STKLOS Procedure

Kills (brutally) *process*. The result of `process-kill` is *void*. This procedure is equivalent to

```
(process-send-signal process 'SIGTERM)
```

process-stop *proc* STKLOS Procedure
process-continue *proc* STKLOS Procedure

process-stop stops the execution of *proc* and **process-continue** resumes its execution. They are equivalent, respectively, to

```
(process-send-signal process 'SIGSTOP)
(process-send-signal process 'SIGCONT)
```

process-list STKLOS Procedure
Returns the list of processes which are currently running (i.e. alive).

6.16 Sockets

STKLOS defines **sockets**, on systems which support them, as first class objects. Sockets permits processes to communicate even if they are on different machines. Sockets are useful for creating client-server applications.

make-client-socket *hostname port-number* STKLOS Procedure
make-client-socket *hostname port-number line-buffered* STKLOS Procedure

make-client-socket returns a new socket object. This socket establishes a link between the running program and the application listening on port *port-number* of *hostname*. If the optional argument *line-buffered* has a true value, a line buffered policy is used when writing to the client socket (i.e. characters on the socket are transmitted as soon as a `#\newline` character is encountered). The default value of *line-buffered* is `#t`.

make-server-socket STKLOS Procedure
make-server-socket *port-number* STKLOS Procedure

make-server-socket returns a new socket object. If *port-number* is specified, the socket is listening on the specified port; otherwise, the communication port is chosen by the system.

socket-shutdown *sock* STKLOS Procedure
socket-shutdown *sock close* STKLOS Procedure

socket-shutdown shutdowns the connection associated to *socket*. If the socket is a server socket, **socket-shutdown** is called on all the clientsockets connected to this server. *close* indicates if the the socket must be closed or not, when the connection is destroyed. Closing the socket forbids further connections on the same port with the **socket-accept** procedure. Omitting a value for *close* implies the closing of socket.

The following example shows a simple server: when there is a new connection on the port number 12345, the server displays the first line sent to it by the client, discards the others and go back waiting for further client connections.

```
(let ((s (make-server-socket 12345)))
  (let loop ()
    (let ((ns (socket-accept s)))
      (format t "I've read: ~A\n" (read-line (socket-input ns)))
      (socket-shutdown ns f)
      (loop))))
```

socket-accept *socket* STKLOS Procedure
socket-accept *socket line-buffered* STKLOS Procedure

socket-accept waits for a client connection on the given *socket*. If no client is already waiting for a connection, this procedure blocks its caller; otherwise, the first connection

request on the queue of pending connections is connected and `socket-accept` returns a new client socket to serve this request. This procedure must be called on a server socket created with `make-server-socket`. The result of `socket-accept` is undefined. `Line-buffered` indicates if the port should be considered as a line buffered. If `line-buffered` is omitted, it defaults to `t`.

The following example is a simple server which waits for a connection on the port 12345⁷. Once the connection with the distant program is established, we read a line on the input port associated to the socket and we write the length of this line on its output port.

```
(let* ((server (make-server-socket 13345))
      (client (socket-accept server))
      (l      (read-line (socket-input client))))
  (format (socket-output client) "Length is: ~a\n" (string-length l))
  (socket-shutdown server))
```

Note that shutting down the server socket suffices here to close also the connection to client.

- socket?** *obj* STKLOS Procedure
Returns `t` if `socket` is a socket, otherwise returns `f`.
- socket-server?** *obj* STKLOS Procedure
Returns `t` if `socket` is a server socket, otherwise returns `f`.
- socket-client?** *obj* STKLOS Procedure
Returns `t` if `socket` is a client socket, otherwise returns `f`.
- socket-host-name** *socket* STKLOS Procedure
Returns a string which contains the name of the distant host attached to `socket`. If `socket` has been created with `make-client-socket` this procedure returns the official name of the distant machine used for connection. If `socket` has been created with `make-server-socket`, this function returns the official name of the client connected to the socket. If no client has used yet `socket`, this function returns `f`.
- socket-host-address** *socket* STKLOS Procedure
Returns a string which contains the IP number of the distant host attached to `socket`. If `socket` has been created with `make-client-socket` this procedure returns the IP number of the distant machine used for connection. If `socket` has been created with `make-server-socket`, this function returns the address of the client connected to the socket. If no client has used yet `socket`, this function returns `f`.
- socket-local-address** *socket* STKLOS Procedure
Returns a string which contains the IP number of the local host attached to `socket`.
- socket-port-number** *socket* STKLOS Procedure
Returns the integer number of the port used for `socket`.
- socket-input** *socket* STKLOS Procedure
socket-output *socket* STKLOS Procedure
Returns the file port associated for reading or writing with the program connected with `socket`. If no connection has already been established, these functions return `f`.

⁷ Under Unix, you can simply connect to a listening socket with the `telnet` command. With the given example, this can be achieved by typing the following command in a window shell: `$ telnet localhost 12345`

The following example shows how to make a client socket. Here we create a socket on port 13 of the machine `kaolin.unice.fr`⁸:

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (format t "Time is: ~A~%" (read-line (socket-input s)))
  (socket-shutdown s))
```

6.17 System Procedures

6.17.1 File Primitives

temporary-file-name STKLOS Procedure

Generates a unique temporary file name. The value returned by `temporary-file-name` is the newly generated name of `#f` if a unique name cannot be generated.

rename-file *string1 string2* STKLOS Procedure

Renames the file whose path-name is `string1` to a file whose path-name is `string2`. The result of `rename-file` is *void*.

remove-file *string* STKLOS Procedure

Removes the file whose path name is given in `string`. The result of `remove-file` is *void*.

copy-file *string1 string2* STKLOS Procedure

Copies the file whose path-name is `string1` to a file whose path-name is `string2`. If the file `string2` already exists, its content prior the call to `copy-file` is lost. The result of `copy-file` is *void*.

file-is-directory? *string* STKLOS Procedure

file-is-regular? *string* STKLOS Procedure

file-is-readable? *string* STKLOS Procedure

file-is-writable? *string* STKLOS Procedure

file-is-executable? *string* STKLOS Procedure

file-exists? *string* STKLOS Procedure

Returns `#t` if the predicate is true for the path name given in `string`; returns `#f` otherwise (or if `string` denotes a file which does not exist).

getcwd STKLOS Procedure

Returns a string containing the current working directory.

chmod *str* STKLOS Procedure

chmod *str option1 ...* STKLOS Procedure

Change the access mode of the file whose path name is given in `string`. The options must be composed of either an integer or one of the following symbols `read`, `write` or `execute`. Giving no option to `chmod` is equivalent to pass it the integer 0. If the operation succeeds, `chmod` returns `#t`; otherwise it returns `#f`.

```
(chmod "~/stklos/stklosrc" 'read 'execute)
(chmod "~/stklos/stklosrc" #o644)
```

⁸ Port 13 is generally used for testing: making a connection to it permits to know the distant system's idea of the time of day.

chdir *dir* STKLOS Procedure
 Changes the current directory to the directory given in string *dir*.

expand-file-name *path* STKLOS Procedure
 Expand-file-name expands the filename given in *path* to an absolute path.

```
;; Current directory is ~eg/STklos (i.e. /users/eg/STklos)
(expand-file-name "..")           => "/users/eg"
(expand-file-name "~eg/./eg/bin") => "/users/eg/bin"
(expand-file-name "~/STklos)"    => "/users/eg/STk"
```

canonical-file-name *path* STKLOS Procedure
 Expands all symbolic links in *path* and returns its canonicalized absolute path name. The resulting path does not have symbolic links. If *path* doesn't designate a valid path name, canonical-file-name returns #f.

decompose-file-name *string* STKLOS Procedure
 Returns an “exploded” list of the path name components given in *string*. The first element in the list denotes if the given *string* is an absolute path or a relative one, being "/" or "." respectively. Each component of this list is a string.

```
(decompose-file-name "/a/b/c.stk") => ("/" "a" "b" "c.stk")
(decompose-file-name "a/b/c.stk")  => ( "." "a" "b" "c.stk")
```

basename *str* STKLOS Procedure
 Returns a string containing the last component of the path name given in *str*.

```
(basename "/a/b/c.stk") => "c.stk"
```

dirname *str* STKLOS Procedure
 Returns a string containing all but the last component of the path name given in *str*.

```
(dirname "/a/b/c.stk") => "/a/b"
```

file-separator STKLOS Procedure
 Returns the operating system file separator as a character.

make-path *dirname name* STKLOS Procedure
 Builds a file name from the directory *dirname* and *name*.

glob *pattern ...* STKLOS Procedure
 Glob performs file name “globbing” in a fashion similar to the csh shell. Glob returns a list of the filenames that match at least one of *pattern* arguments. The *pattern* arguments may contain the following special characters:

- ? Matches any single character.
- * Matches any sequence of zero or more characters.
- [chars] Matches any single character in *chars*. If *chars* contains a sequence of the form *a-b* then any character between *a* and *b* (inclusive) will match.
- \\ Matches the character *x*.
- {*a,b,...*} Matches any of the strings *a*, *b*, etc.

As with `cs`, a `'/'` at the beginning of a file's name or just after a `'/'` must be matched explicitly or with a `{}` construct. In addition, all `'/'` characters must be matched explicitly. If the first character in a pattern is `'~'` then it refers to the home directory of the user whose name follows the `'~'`. If the `'~'` is followed immediately by `'/'` then the value of the environment variable `HOME` is used.

`Glob` differs from `cs` globbing in two ways. First, it does not sort its result list (use the `sort` procedure if you want the list sorted). Second, `glob` only returns the names of files that actually exist; in `cs` no check for existence is made unless a pattern contains a `?`, `*`, or `{}` construct.

6.17.2 Environment

getenv *str* STKLOS Procedure
getenv STKLOS Procedure

Looks for the environment variable named `str` and returns its value as a string, if it exists. Otherwise, `getenv` returns `#f`. If `getenv` is called without parameter, it returns the list of all the environment variables accessible from the program as an A-list.

```
(getenv "SHELL")
⇒ "/bin/zsh"
(getenv)
⇒ (("TERM" . "xterm") ("PATH" . "/bin:/usr/bin") ...)
```

setenv! *var value* STKLOS Procedure
Sets the environment variable `var` to `value`. `Var` and `value` must be strings. The result of `setenv!` is *void*.

6.17.3 System Informations

running-os STKLOS Procedure
Returns the name of the underlying Operating System which is running the program. The value returned by `running-os` is a symbol. For now, this procedure returns either `unix` or `windows`.

hostname STKLOS Procedure
Return the host name of the current processor as a string.

argc STKLOS Procedure
Returns the number of argument present on the command line

argv STKLOS Procedure
Returns a list of the arguments given on the shell command line. The interpreter options are no included in the result

program-name STKLOS Procedure
Returns the invocation name of the current program as a string.

version STKLOS Procedure
Returns a string identifying the current version of the system. A version is constituted of three numbers separated by a point: the version, the release and sub-release numbers.

- machine-type** STKLOS Procedure
Returns a string identifying the kind of machine which is running the program. The result string is of the form `[os-name]-[os-version]-[processor-type]`.
- clock** STKLOS Procedure
Returns an approximation of processor time, in milliseconds, used so far by the program.
- date** STKLOS Procedure
Returns the current date in a string
- current-time** STKLOS Procedure
Returns the time since the Epoch (that is 00:00:00 UTC, January 1, 1970), measured in seconds.
- full-current-time** STKLOS Procedure
Returns the time of the day as a pair where
- the first element is the time since the Epoch (that is 00:00:00 UTC, January 1, 1970), measured in seconds.
 - the second element is the number of microseconds in the given second.
- seconds->date** *sec* STKLOS Procedure
Returns a keyword list for the date given by *sec* (a date based on the Epoch). The keyed values returned are
- `second` : 0 to 59 (but can be up to 61 to allow for leap seconds)
 - `minute` : 0 to 59
 - `hour` : 0 to 23
 - `day` : 1 to 31
 - `month` : 1 to 12
 - `year` : e.g., 2002
 - `week-day` : 0 (Sunday) to 6 (Saturday)
 - `year-day` : 0 to 365 (365 in leap years)
 - `dst?` : `#t` (daylight savings time) or `#f`
 - `time-zone-offset` : the difference between Coordinated Universal Time (UTC) and local standard time in seconds.
- Example:
- ```
(seconds->date (current-time)) ⇒ (:second 49 :minute 32 :hour 23
 :day 2 :month 4 :year 2002
 :week-day 2 :year-day 91
 :dst #t :time-zone-offset -3600)
```
- time** *expr1 expr2 ...* STKLOS Syntax  
Evaluates the expressions *expr1*, *expr2*, ... and returns the result of the last expression. This form prints also the time spent for this evaluation on the current error port.
- getpid** STKLOS Procedure  
Returns the system process number of the current program (i.e. the Unix *pid*) as an integer.

### 6.17.4 Program Arguments Parsing

STKLOS provides a simple way to parse program arguments with the `parse-arguments` special form. This form is generally used into the `main` function in a Scheme script. See SRFI-22 see [Appendix B \[SRFIs\], page 78](#) on how to use a `main` function in a Scheme program.

**parse-arguments** *<args>* *<clause1>* *<clause2>* ... STKLOS Procedure

The `parse-arguments` special form is used to parse the command line arguments of a Scheme script. The implementation of this form internally uses the GNU C `getopt` function. As a consequence `parse-arguments` accepts options which start with the `'` (short option) or `-` characters (long option).

The first argument of `parse-arguments` is a list of the arguments given to the program (comprising the program name in the CAR of this list). Following arguments are clauses. Clauses are described later.

By default, `parse-arguments` permutes the contents of (a copy) of the arguments as it scans, so that eventually all the non-options are at the end. However, if the shell environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered.

A clause must follow the syntax:

```

<clause> ⇒ string | <list-clause>
<list clause> ⇒ (<option descr> <expr> ...) | (else <expr> ...)
<option descr> ⇒ (<option name> [<keyword> value]*)
<option name> ⇒ string
<keyword> ⇒ :alternate | :arg | :help

```

A string clause is used to build the help associated to the command. A list clause must follow the syntax describes an option. The `<expr>`s associated to a list clauses are executed when the option is recognized. The `else` clauses is executed when all parameters have been parsed. The `:alternate` key permits to have an alternate name for an option (generally a short or long name if the option name is a short or long name). The `:help` is used to provide help about the the option. The `:arg` is used when the option admit a parameter: the symbol given after `:arg` will be bound to the value of the option argument when the corresponding `<expr>`s will be executed.

In an `else` clause the symbol `other-arguments` is bound to the list of the arguments which are not options.

The following example shows a rather complete usage of the `parse-arguments` form

```

#!/usr/bin/env stklos-script

(define (main args)
 (parse-arguments args
 "Usage: foo [options] [parameter ...]"
 "General options:"
 (("verbose" :alternate "v" :help "be more verbose")
 (format #t "Seen the verbose option\n"))
 (("long" :help "a long option alone")
 (format #t "Seen the long option\n"))
 (("s" :help "a short option alone")
 (format #t "Seen the short option\n"))
 "File options:"
 (("input" :alternate "f" :arg file :help "use <file> as input")
 (format #t "Seen the input option with ~S argument\n" file))

```

```

 ("output" :alternate "o" :arg file :help "use <file> as output")
 (format #t "Seen the output option with ~S argument\n" file))
 "Misc:"
 ("help" :alternate "h" :help "provides help for the command")
 (arg-usage (current-error-port))
 (exit 1))
 (else
 (format #t "All options parsed. Remaining arguments are ~S\n"
 other-arguments))))

```

The following program invocation

```
foo -vs --input in -o out arg1 arg2
```

produces the following output

```

Seen the verbose option
Seen the short option
Seen the input option with "in" argument
Seen the output option with "out" argument
All options parsed. Remaining arguments are ("arg1" "arg2")

```

Finally, the program invocation

```
foo --help
```

produces the following output

```

Usage: foo [options] [parameter ...]
General options:
 --verbose, -v be more verbose
 --long a long option alone
 -s a short option alone
File options:
 --input=<file>, -f <file> use <file> as input
 --output=<file>, -o <file> use <file> as output
Misc:
 --help, -h provides help for the command

```

*Note:*

- Short option can be concatenated. That is,
 

```
prog -abc
```

 is equivalent to the following program call
 

```
prog -a -b -c
```
- Any argument following a '-' argument is no more considered as an option, even if it starts with a '-' or '-'.
- Option with a parameter can be written in several ways. For instance to set the output in the bar file for the previous example can be expressed as
 

```

--output=bar
-o bar
-obar

```

**arg-usage** *port*

STKLOS Procedure

**arg-usage** *as-sexpr*

STKLOS Procedure

This procedure is only bound inside a `parse-arguments` form. It pretty prints the help associated to the clauses of the `parse-arguments` form on the given port. If the argument `as-sexpr` is passed and is not `#f`, the help strings are printed on `port` as *S-exprs*. This is useful if the help strings need to be manipulated by a program.

## 6.17.5 Misc.

**system** *string* STKLOS Procedure

Sends the given **string** to the system shell `/bin/sh`. The result of **system** is the integer status code the shell returns.

**exec** *str* STKLOS Procedure

**exec-list** *str* STKLOS Procedure

These procedures execute the command given in **str**. The command given in **str** is passed to `/bin/sh`. **Exec** returns a strings which contains all the characters that the command **str** has printed on it's standard output, whereas **exec-list** returns a list of the lines which constitute the output of **str**.

```
(exec "echo A; echo B") ⇒ "A\nB\n"
(exec-list "echo A; echo B") ⇒ ("A" "B")
```

**register-exit-function!** *proc* STKLOS Procedure

This function registers **proc** as an exit function. This function will be called when the program exits. When called, **proc** will be passed one parmater which is the status given to the **exit** function. The result of **register-exit-function!** is undefined.

```
(let* ((tmp (temporary-file-name))
 (out (open-output-file tmp)))
 (register-exit-function! (lambda (n)
 (when (zero? n)
 (remove-file tmp))))
 out)
```

**exit** STKLOS Procedure

**exit** *ret-code* STKLOS Procedure

Exits the program with the specified integer return code. If **ret-code** is omitted, the program terminates with a return code of 0. If program has registered exit functions with **register-exit-function!**, they are called (in an order which is the reverse of their call order).

**die** *message* STKLOS Procedure

**die** *message status* STKLOS Procedure

**Die** prints the given **message** on the current error port and exits the program with the **status** value. If **status** is omitted, it defaults to 1.

**address-of** *obj* *R<sup>5</sup>RS* Procedure

Returns the address of the object **obj** as an integer.

**gc** *R<sup>5</sup>RS* Procedure

Returns the address of the object **obj** as an integer.

**void** STKLOS Procedure

**void** *arg1 ...* STKLOS Procedure

Returns the special **void** object. If arguments are passed to **void**, they are evaluated and simply ignored.

**error** *str obj ...* STKLOS Procedure  
**error** *name str obj ...* STKLOS Procedure

**error** is used to signal an error to the user. The second form of **error** takes a symbol as first parameter; it is generally used for the name of the procedure which raises the error.

**Note:** The specification string may follow the “*tilde conventions*” of **format** (see [format], page 48); in this case this procedure builds an error message according to the specification given in *str*. Otherwise, this procedure is conform to the **error** procedure defined in SRFI-23 and *str* is printed with the **display** procedure, whereas the *objs* are printed with the **write** procedure.

Hereafter, are some calls of the **error** procedure using a formatted string

```
(error "bad integer ~A" "a")
 ↪ bad integer a
(error 'vector-ref "bad integer ~S" "a")
 ↪ vector-ref: bad integer "a"
(error 'foo "~A is not between ~A and ~A" "bar" 0 5)
 ↪ foo: bar is not between 0 and 5
```

and some conform to SRFI-23

```
(error "bad integer" "a")
 ↪ bad integer "a"
(error 'vector-ref "bad integer" "a")
 ↪ vector-ref: bad integer "a"
(error "bar" "is not between" 0 "and" 5)
 ↪ bar "is not between" 0 "and" 5
```

**apropos** *obj* STKLOS Procedure  
**apropos** *obj module* STKLOS Procedure

**apropos** returns a list of symbols whose print name contains the characters of *obj* as a substring. The given *obj* can be a string or symbol. This function returns the list of matched symbols which can be accessed from the given *module* (defaults to the current module if not provided).

**trace** *f-name ...* STKLOS Syntax

Invoking **trace** with one or more function names causes the functions named to be traced. Henceforth, whenever such a function is invoked, information about the call and the returned values, if any, will be printed on the current error port.

Calling **trace** with no argument returns the list of traced functions.

**untrace** *f-name ...* STKLOS Syntax

Invoking **untrace** with one or more function names causes the functions named not to be traced anymore.

Calling **untrace** with no argument will print the list of traced functions on the current error port.

**pretty-print** *sexpr :key port width* STKLOS Procedure  
**pp** *sexpr :key port width* STKLOS Procedure

This function tries to obtain a pretty-printed representation of *sexpr*. The pretty-printed form is written on *port* with lines which are no more long than *width* characters. If *port* is omitted it defaults to the current error port. As a special convention, if *port* is **#t**, output goes to the current output port and if *port* is **#f**, the output is returned as a string by **pretty-print**. Note that **pp** is another name for **pretty-print**.

**uri-parse** *str*

STKLOS Procedure

Parses the string *str* as a RFC-2396 URI and return a keyed list with the following components

```

scheme : the scheme used as a string (defaults to "file")
host : the host as a string (defaults to "")
port : the port as an integer (0 if no port specified)
path : the path
query : the query part of the URI as a string (defaults to the empty string)
fragment : the fragment of the URI as a string (defaults to the empty string)

(uri-parse "http://google.com")
⇒ (:scheme "http" :host "google.com" :port 80 :path "/"
 :query "" :fragment "")
(uri-parse "http://stklos.net:8080/a/file?x=1;y=2#end")
⇒ (:scheme "http" :host "stklos.net" :port 8080
 :path "/a/file" :query "x=1;y=2" :fragment "end")
(uri-parse "/a/file")
⇒ (:scheme "file" :host "" :port 0 :path "/a/file"
 :query "" :fragment "")
(uri-parse "")
⇒ (:scheme "file" :host "" :port 0 :path ""
 :query "" :fragment "")

```

**string->html** *str*

STKLOS Procedure

This primitive is a convenience function; it returns a string where the HTML special chars are properly translated. It can easily be written in Scheme, but this version is fast.

```

(string->html "Just a <test>")
⇒ "Just a <test>"

```

## 6.18 Signals

To be written

## 6.19 Parameter Objects

STKLOS parameters correspond to the ones defined in SRFI-39. See SRFI document for more information.

**make-parameter** *init*

STKLOS Procedure

**make-parameter** *init converter*

STKLOS Procedure

Returns a new parameter object which is bound in the global dynamic environment to a cell containing the value returned by the call (`converter init`). If the conversion procedure `converter` is not specified the identity function is used instead.

The parameter object is a procedure which accepts zero or one argument. When it is called with no argument, the content of the cell bound to this parameter object in the current dynamic environment is returned. When it is called with one argument, the content of the cell bound to this parameter object in the current dynamic environment is set to the result of the call (`converter arg`), where `arg` is the argument passed to the parameter object, and an unspecified value is returned.

```

(define radix
 (make-parameter 10))

(define write-shared
 (make-parameter
 #f
 (lambda (x)
 (if (boolean? x)
 x
 (error "only booleans are accepted by write-shared")))))

(radix) ⇒ 10
(radix 2)
(radix) ⇒ 2
(write-shared 0) gives an error

(define prompt
 (make-parameter
 123
 (lambda (x)
 (if (string? x)
 x
 (with-output-to-string (lambda () (write x)))))))

(prompt) ⇒ "123"
(prompt ">")
(prompt) ⇒ ">"

```

**parameterize** *expr1 expr2 ... <body>*

STKLOS Syntax

The expressions *expr1* and *expr2* are evaluated in an unspecified order. The value of the *expr1* expressions must be parameter objects. For each *expr1* expression and in an unspecified order, the local dynamic environment is extended with a binding of the parameter object *expr1* to a new cell whose content is the result of the call (`converter val`), where *val* is the value of *expr2* and *converter* is the conversion procedure of the parameter object. The resulting dynamic environment is then used for the evaluation of *<body>* (which refers to the  $R^5RS$  grammar nonterminal of that name). The result(s) of the *parameterize* form are the result(s) of the *<body>*.

```

(radix) ⇒ 2
(parameterize ((radix 16)) (radix)) ⇒ 16
(radix) ⇒ 2

(define (f n) (number->string n (radix)))

(f 10) ⇒ "1010"
(parameterize ((radix 8)) (f 10)) ⇒ "12"
(parameterize ((radix 8) (prompt (f 10))) (prompt)) ⇒ "1010"

```

**parameter?** *obj*

STKLOS Procedure

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

## 6.20 Customization

**real-precision**

STKLOS Procedure

**real-precision** *value*

STKLOS Procedure

This parameter object permits to change the default precision used to print real numbers.

```
(real-precision) ⇒ 15
(define f 0.123456789)
(display f) ⇨ 0.123456789
(real-precision 3)
(display f) ⇨ 0.123
```

## 7 STklos Object System

## Appendix A Using the SLIB package

Aubrey Jaffer maintains a package called *SLIB* which is a portable Scheme library which provides compatibility and utility functions for all standard Scheme implementations. To use this package, you have just to type

```
(require "slib")
```

and follow the instructions given in the *SLIB* library to use a particular package.

**Note:** *SLIB* uses also the *require/provide* mechanism to load components of the library. Once *SLIB* has been loaded, the standard STKLOS `require` and `provide` are overloaded such as if their parameter is a string this is the old STKLOS procedure which is called, and if their parameter is a symbol, this is the *SLIB* one which is called.

## Appendix B SRFIs

The *Scheme Request for Implementation* (SRFI) process grew out of the Scheme Workshop held in Baltimore, MD, on September 26, 1998, where the attendees considered a number of proposals for standardized feature sets for inclusion in Scheme implementations. Many of the proposals received overwhelming support in a series of straw votes. Along with this there was concern that the next Revised Report would not be produced for several years and this would prevent the timely implementation of standardized approaches to several important problems and needs in the Scheme community.

Only the implemented SRFIs are (briefly) presented here. For further information on each SRFI, please look at the official SRFI site <http://srfi.schemers.org>

### SRFI-0 – Conditional Expansion

This SRFI defines the `cond-expand` special form. It is fully supported by STKLOS. STKLOS defines several features identifiers which are of the form *srfi-n* where *n* represents the number of the SRFI supported by the implementation (for instance *srfi-1* or *srfi-30*). Furthermore, the feature identifier *stklos* is defined for application which need to know on which Scheme implementation they are running on.

### SRFI 1 – List Primitives

This SRFI defines an extensive library for list manipulation. The implementation used in STklos is based on the reference implementation from Olin Shivers. To use, SRFI-1 you need to insert the following expression

```
(require "srfi-1")
```

in your code or uses the `cond-expand` special form.

### SRFI 2 – `and-let*` special-form

This SRFI defines an *and* form with local binding which acts as a guarded *let\**. To use, SRFI-2 you need to insert the following expression

```
(require "srfi-2")
```

in your code or uses the `cond-expand` special form.

## SRFI 4 – Homogeneous Numeric Vectors

This SRFI defines a set of data types for vectors whose element are of the same numeric type (homogeneous vectors). To use SRFI-4, you need to insert the following expression

```
(require "srfi-4")
```

in your code or uses the `cond-expand` special form.

## SRFI 6 – String Ports

This SRFI is fully supported and is completely described in this document (procedures `open-input-string`, `open-output-string` and `get-output-string`).

## SRFI 7 – Feature-based program configuration language

This SRFI is fully supported. To use SRFI-7, you need to insert the following expression

```
(require "srfi-7")
```

in your code or uses the `cond-expand` special form.

## SRFI 8 – Binding to Multiple Values

This SRFI is fully supported and is completely described in this document (special form `receive`)

## SRFI 9 – Defining Record Types

This SRFI is fully supported (the implementation uses `STKLOS` classes to implement SRFI-9 records). To use SRFI-9, you need to insert the following expression

```
(require "srfi-9")
```

in your code or uses the `cond-expand` special form.

## SRFI 11 – `let-values` and `let*-values`

This SRFI is fully supported. To use SRFI-11, you need to insert the following expression

```
(require "srfi-11")
```

in your code or uses the `cond-expand` special form.

## SRFI 13 – String Libraries

This SRFI is fully supported. To use SRFI-13, you need to insert the following expression

```
(require "srfi-13")
```

in your code or uses the `cond-expand` special form.

## SRFI 14 – Characters sets

This SRFI is fully supported. To use SRFI-14, you need to insert the following expression

```
(require "srfi-14")
```

in your code or uses the `cond-expand` special form.

## SRFI 16 – Syntax for procedures of variable arity

This SRFI is fully supported and is completely described in this document (procedure `case-lambda`).

## SRFI 19 – Time Data Types and Procedures

This SRFI is fully supported. To use SRFI-19, you need to insert the following expression

```
(require "srfi-19")
```

in your code or uses the `cond-expand` special form.

## SRFI 22 – Unix Scheme Scripts

This SRFI describes basic prerequisites for running Scheme programs as Unix scripts in a uniform way. Specifically, it describes:

- the syntax of Unix scripts written in Scheme,
- a uniform convention for calling the Scheme script interpreter, and
- a method for accessing the Unix command line arguments from within the Scheme script.

SRFI-22 recommends to invoke the Scheme script interpreter from the script via a `/usr/bin/env` trampoline, like this: `#!/usr/bin/env <executable>` where `<executable>` can recover several specified names. STKLOS uses only the name `stklos-script` for `<executable>`.

Here is an example of the classical `echo` command (without option) in Scheme:

```
#!/usr/bin/env stklos-script

(define (main arguments)
 (for-each (lambda (x) (display x) (display #\space))
 (cdr arguments))
 (newline)
 0)
```

## SRFI 23 – Error reporting mechanism

This SRFI is fully supported. See the documentation of the `error` primitive form more information (in fact STKLOS `error` is more general than the one defined in SRFI-23).

## SRFI 26 – Specializing Parameters without Currying

This SRFI is fully supported. To use SRFI-31, you need to insert the following expression

```
(require "srfi-26")
```

in your code or uses the `cond-expand` special form.

## SRFI 27 – Sources of Random Bits

This SRFI is fully supported.

## SRFI 28 – Basic Format Strings

This SRFI is fully supported. See the documentation of the `format` primitive form more information (in fact STKLOS `format` is more general than the one defined in SRFI-28).

## **SRFI 30 – Nested Multi-line Comments**

This SRFI is fully supported by STKLOS reader.

## **SRFI 31 – A special form for recursive evaluation**

This SRFI is fully supported. To use SRFI-31, you need to insert the following expression

```
(require "srfi-31")
```

in your code or uses the `cond-expand` special form.

## **SRFI 38 – External Representation for Data With Shared Structure**

This SRFI is fully supported by STKLOS reader.

## **SRFI 39 – Parameter objects**

This SRFI is fully supported and is completely described in this document (procedures `make-parameter` and `parameterize`).

## Appendix C Bibliography

- [Gallesio95] Erick Gallesio. ‘STk Reference Manual.’ I3S CNRS / Université de Nice - Sophia Antipolis, RT95-31a, juillet 1995,
- [GTK01] ‘The GTK+ Graphical Toolkit’ available from <http://gtk.org>
- [PCRE01] P. Hazel. ‘The Perl Compatible Regular Expression Package.’ available from <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>
- [QueinnecGeffroy92] C. Queinnec and J-M. Geffroy. ‘Partial evaluation applied to symbolic pattern matching with intelligent backtrack.’ In M. Billaud, P. Casteran, MM. Corsini, K. Musumbu, and A. Rauzy: Editors, *Workshop on Static Analysis*, number 81-82 in bigre, pages 109–117, Bordeaux (France), September 1992.
- [R5RS] R Kelsey, W. Clinger and J. Rees: Editors. ‘The Revised(5) Report on the Algorithmic Language Scheme’.
- [Serrano01] M. Serrano. ‘Bigloo User’s Manual, v2.4.’ INRIA, november 2001.
- [Ousterhout91] John K. Ousterhout. ‘An X11 Toolkit Based on the Tcl Language.’ USENIX Winter Conference, January 1991, pages 105-115
- [DSSSL96] ISO/IEC. Information technology, Processing Languages, ‘Document Style Semantics and Specification Languages (DSSSL)’ Technical Report 10179 :1996(E), ISO, 1996.

# Index

## #

#! comment for Scheme scripts ..... 1  
 #| multi-line comment ..... 1

,

'<datum> ..... 2

\*

\* ..... 20

,

, in quasiquote ..... 10

,@ in quasiquote ..... 10

-

- ..... 20

.

..... 26

/

/ ..... 20

/bin/sh ..... 72

:

:key lambda parameter ..... 3

:optional lambda parameter ..... 3

:rest lambda parameter ..... 3

=

= ..... 20

‘

‘<template> ..... 10

+

+ ..... 20

>

> ..... 20

>= ..... 20

<

< ..... 20

<= ..... 20

## A

abs ..... 20

acos ..... 23

address-of ..... 72

all-modules ..... 16

and ..... 6

angle ..... 23

any ..... 39

append ..... 27

append! ..... 27

apply ..... 38

apropos ..... 73

arg-usage ..... 71

argc ..... 68

argv ..... 68

ASCII ..... 31, 52

asin ..... 23

assoc ..... 28

assq ..... 28

assv ..... 28

atan ..... 23

## B

Backquote ..... 10

basename ..... 67

begin ..... 9

Bibliography ..... 82

Binding constructs ..... 7

bit-and ..... 24

bit-not ..... 24

bit-or ..... 24

bit-shift ..... 24

bit-xor ..... 24

Boolean value ..... 25

boolean? ..... 25

## C

caar ..... 26

cadr ..... 26

Call by need ..... 10

call-with-current-continuation ..... 40

call-with-input-file ..... 42

call-with-input-string ..... 42

call-with-output-file ..... 42

call-with-output-string ..... 42

call-with-values ..... 40

call/cc ..... 40

canonical-file-name ..... 67

|                     |    |
|---------------------|----|
| car                 | 25 |
| case                | 5  |
| case-lambda         | 4  |
| case-lambda         | 80 |
| cdddar              | 26 |
| cddddr              | 26 |
| cdr                 | 26 |
| ceiling             | 22 |
| char->integer       | 32 |
| char-alphabetic?    | 32 |
| char-ci=?           | 32 |
| char-ci>=?          | 32 |
| char-ci>?           | 32 |
| char-ci<=?          | 32 |
| char-ci<?           | 32 |
| char-downcase       | 32 |
| char-lower-case?    | 32 |
| char-numeric?       | 32 |
| char-ready?         | 46 |
| char-upcase         | 32 |
| char-upper-case?    | 32 |
| char-whitespace?    | 32 |
| char=?              | 32 |
| char?               | 31 |
| char>=?             | 32 |
| char>?              | 32 |
| char<=?             | 32 |
| char<?              | 32 |
| Character           | 31 |
| Character sets      | 79 |
| chdir               | 67 |
| chmod               | 66 |
| Circular structure  | 2  |
| Class               | 76 |
| clock               | 69 |
| CLOS                | 76 |
| close-input-port    | 45 |
| close-output-port   | 45 |
| close-port          | 45 |
| Closure             | 3  |
| closure?            | 4  |
| Comments            | 1  |
| complex?            | 19 |
| cond                | 5  |
| Conditional         | 5  |
| cons                | 25 |
| copy-file           | 66 |
| copy-tree           | 29 |
| cos                 | 23 |
| current-error-port  | 43 |
| current-input-port  | 43 |
| current-module      | 14 |
| current-output-port | 43 |
| current-time        | 69 |

**D**

|                             |    |
|-----------------------------|----|
| date                        | 69 |
| decompose-file-name         | 67 |
| define-macro                | 11 |
| define-module               | 14 |
| define-syntax               | 12 |
| delay                       | 10 |
| delete                      | 29 |
| delete!                     | 29 |
| denominator                 | 21 |
| die                         | 72 |
| Different kinds of comments | 1  |
| dirname                     | 67 |
| display                     | 47 |
| do                          | 9  |
| dotimes                     | 10 |
| dynamic-wind                | 41 |

**E**

|                  |        |
|------------------|--------|
| eof-object?      | 46     |
| eq?              | 18     |
| equal?           | 19     |
| equiv?           | 17     |
| error            | 73, 80 |
| eval             | 41     |
| even?            | 20     |
| every            | 39     |
| exact->inexact   | 23     |
| exact?           | 19     |
| exec             | 72     |
| exec-list        | 72     |
| exit             | 72     |
| exp              | 22     |
| expand-file-name | 67     |
| export           | 15     |
| expt             | 23     |

**F**

|                     |        |
|---------------------|--------|
| False value         | 25     |
| file-exists?        | 66     |
| file-is-directory?  | 66     |
| file-is-executable? | 66     |
| file-is-readable?   | 66     |
| file-is-regular?    | 66     |
| file-is-writable?   | 66     |
| file-separator      | 67     |
| filter              | 29     |
| filter!             | 29     |
| find-module         | 14     |
| find-path           | 49     |
| floor               | 22     |
| fluid-let           | 8      |
| flush               | 48     |
| for-each            | 39     |
| force               | 39     |
| format              | 48, 80 |

full-current-time ..... 69

## G

gc ..... 72  
gcd ..... 21  
gensym ..... 31  
get-output-string ..... 45, 79  
getcwd ..... 66  
getenv ..... 68  
getpid ..... 69  
glob ..... 67  
Global Variable ..... 14  
GTK+ ..... 1

## H

Hash Table ..... 59  
hash-table->list ..... 62  
hash-table-for-each ..... 61  
hash-table-get ..... 61  
hash-table-hash ..... 60  
hash-table-map ..... 62  
hash-table-put! ..... 61  
hash-table-remove ..... 61  
hash-table-stats ..... 62  
hash-table-update! ..... 61  
hash-table? ..... 60  
hostname ..... 68  
Hygienic Macros ..... 11

## I

if ..... 5  
imag-part ..... 23  
import ..... 15  
in-module ..... 16  
inexact->exact ..... 23  
inexact? ..... 19  
Input ..... 41  
input-file-port? ..... 43  
input-port? ..... 42  
input-string-port? ..... 43  
integer->char ..... 32  
integer? ..... 19  
interactive-port? ..... 43

## K

key-delete! ..... 50  
key-get ..... 50  
key-set! ..... 50  
keyword ..... 1  
keyword ..... 49  
Keyword ..... 49  
Keyword procedure parameter ..... 3  
keyword->string ..... 50

## L

lambda ..... 3  
last-pair ..... 28  
Lazy evaluation ..... 10  
lcm ..... 21  
length ..... 27  
let ..... 7  
let\* ..... 7  
let\*-values ..... 79  
let-syntax ..... 12  
let-values ..... 79  
letrec ..... 8  
letrec-syntax ..... 13  
list ..... 26  
List ..... 25  
list\* ..... 27  
list->string ..... 34  
list->vector ..... 37  
list-ref ..... 28  
list-tail ..... 28  
list? ..... 26  
load ..... 48  
load-path ..... 48  
load-suffixes ..... 49  
log ..... 23  
Low Level Macros ..... 11

## M

machine-type ..... 69  
Macro Expansion ..... 13  
macro-expand ..... 13  
Macros ..... 11  
Macros, Hygienic ..... 11  
Macros, Low Level ..... 11  
Macros, Referentially Transparent ..... 11  
magnitude ..... 23  
make-client-socket ..... 64  
make-hash-table ..... 60  
make-keyword ..... 50  
make-parameter ..... 74  
make-path ..... 67  
make-polar ..... 23  
make-rectangular ..... 23  
make-server-socket ..... 64  
make-string ..... 33  
make-vector ..... 36  
map ..... 38  
match-case ..... 58  
match-lambda ..... 58  
max ..... 20  
member ..... 28  
memq ..... 28  
memv ..... 28  
min ..... 20  
module-exports ..... 16  
module-imports ..... 16  
module-name ..... 16

module-symbols ..... 16  
 module? ..... 15  
 Modules ..... 14  
 modulo ..... 21  
 Multiple Values ..... 40

## N

Name Space ..... 14  
 negative? ..... 20  
 newline ..... 47  
 not ..... 25  
 null? ..... 26  
 number->string ..... 23  
 number? ..... 19  
 numerator ..... 21

## O

Object ..... 76  
 odd? ..... 20  
 open-file ..... 44  
 open-input-file ..... 44  
 open-input-string ..... 44, 79  
 open-output-file ..... 44  
 open-output-string ..... 44, 79  
 Optional procedure parameter ..... 3  
 or ..... 6  
 Output ..... 41  
 output-file-port? ..... 43  
 output-port? ..... 42  
 output-string-port? ..... 43

## P

Pair ..... 25  
 pair-mutable? ..... 26  
 pair? ..... 25  
 parameter? ..... 75  
 parameterize ..... 75  
 parse-arguments ..... 70  
 Pattern Matching ..... 58  
 peek-char ..... 46  
 PID ..... 62  
 port->sexp-list ..... 47  
 port->string ..... 47  
 port->string-list ..... 47  
 port-current-line ..... 45  
 port-file-name ..... 45  
 port-idle-register! ..... 45  
 port-idle-reset! ..... 45  
 port-idle-unregister! ..... 45  
 positive? ..... 20  
 Posix ..... 53  
 pp ..... 73  
 pretty-print ..... 73  
 Procedure ..... 3  
 procedure? ..... 38

procedures of variable arity ..... 80  
 Process ..... 62  
 process-alive? ..... 63  
 process-continue ..... 64  
 process-exit-status ..... 63  
 process-input ..... 63  
 process-kill ..... 63  
 process-list ..... 64  
 process-pid ..... 63  
 process-send-signal ..... 63  
 process-stop ..... 64  
 process-wait ..... 63  
 process? ..... 63  
 program-name ..... 68  
 promise ..... 10  
 promise? ..... 10  
 provide ..... 49, 77  
 provided? ..... 49

## Q

quasiquote ..... 10  
 Quasiquote ..... 10  
 quote ..... 2  
 quotient ..... 21

## R

R5RS ..... 1  
 random-integer ..... 24, 80  
 random-real ..... 25, 80  
 rational? ..... 19  
 rationalize ..... 22  
 read ..... 45  
 read-char ..... 46  
 read-line ..... 46  
 read-with-shared-structure ..... 46  
 real-part ..... 23  
 real-precision ..... 76  
 real? ..... 19  
 receive ..... 41, 79  
 regexp-match ..... 56  
 regexp-match-positions ..... 56  
 regexp-quote ..... 57  
 regexp-replace ..... 57  
 regexp-replace-all ..... 57  
 regexp? ..... 56  
 register-exit-function! ..... 72  
 Regular Expression ..... 50  
 remainder ..... 21  
 remove ..... 29  
 remove-file ..... 66  
 rename-file ..... 66  
 require ..... 49, 77  
 reverse ..... 27  
 reverse! ..... 27  
 rewind-file-port ..... 45  
 round ..... 22

run-process ..... 62  
 running-os ..... 68

## S

Script Files ..... 1, 80  
 seconds->date ..... 69  
 select-module ..... 15  
 set-car! ..... 26  
 set-cdr! ..... 26  
 set-load-path! ..... 49  
 set-load-suffixes! ..... 49  
 setenv! ..... 68  
 Shell variable, STKLOS\_LOAD\_PATH ..... 49  
 Signal ..... 74  
 sin ..... 23  
 SLIB ..... 11, 77  
 socket-accept ..... 64  
 socket-client? ..... 65  
 socket-host-address ..... 65  
 socket-host-name ..... 65  
 socket-input ..... 65  
 socket-local-address ..... 65  
 socket-output ..... 65  
 socket-port-number ..... 65  
 socket-server? ..... 65  
 socket-shutdown ..... 64  
 socket? ..... 65  
 Sockets ..... 64  
 sort ..... 37  
 sqrt ..... 23  
 SRFI ..... 78  
 SRFI-13 ..... 35  
 SRFI-22 ..... 70  
 SRFI-23 ..... 72  
 SRFI-28 ..... 48  
 SRFI-38 ..... 46, 47  
 SRFI-6 ..... 44, 45  
 STk ..... 1  
 STKLOS\_LOAD\_PATH ..... 49  
 string ..... 33  
 String ..... 33  
 String Libraries ..... 79  
 String Port ..... 41  
 string->html ..... 74  
 string->list ..... 34  
 string->number ..... 24  
 string->regexp ..... 56  
 string->symbol ..... 30  
 string->unterned-symbol ..... 31  
 string-append ..... 34  
 string-ci=? ..... 34  
 string-ci>=? ..... 34  
 string-ci>? ..... 34  
 string-ci<=? ..... 34  
 string-ci<? ..... 34  
 string-copy ..... 34  
 string-downcase ..... 35

string-downcase! ..... 35  
 string-fill! ..... 35  
 string-find? ..... 35  
 string-index ..... 34  
 string-length ..... 33  
 string-mutable? ..... 36  
 string-ref ..... 33  
 string-set! ..... 33  
 string-split ..... 34  
 string-titlecase ..... 35  
 string-titlecase! ..... 36  
 string-upcase ..... 35  
 string-upcase! ..... 35  
 string=? ..... 34  
 string? ..... 33  
 string>=? ..... 34  
 string>? ..... 34  
 string<=? ..... 34  
 string<? ..... 34  
 substring ..... 34  
 sxhash Common Lisp Function ..... 60  
 symbol->string ..... 30  
 symbol-value ..... 16  
 symbol-value\* ..... 16  
 symbol? ..... 30  
 syntax-rules ..... 12  
 system ..... 72

## T

tan ..... 23  
 temporary-file-name ..... 66  
 the pattern language ..... 59  
 tilde expansion ..... 67  
 time ..... 69  
 Tk ..... 1  
 trace ..... 73  
 True value ..... 25  
 truncate ..... 22  
 try-load ..... 48

## U

unless ..... 6  
 unquote ..... 10  
 unquote-splicing ..... 10  
 until ..... 10  
 untrace ..... 73  
 uri-parse ..... 74

**V**

|                        |    |
|------------------------|----|
| values .....           | 40 |
| Variable, Global ..... | 14 |
| vector .....           | 36 |
| vector->list .....     | 37 |
| vector-copy .....      | 37 |
| vector-fill! .....     | 37 |
| vector-length .....    | 36 |
| vector-mutable? .....  | 37 |
| vector-ref .....       | 37 |
| vector-resize .....    | 37 |
| vector-set! .....      | 37 |
| vector? .....          | 36 |
| Vectors .....          | 36 |
| version .....          | 68 |
| void .....             | 72 |

**W**

|                                   |    |
|-----------------------------------|----|
| when .....                        | 6  |
| while .....                       | 10 |
| with-error-to-file .....          | 43 |
| with-input-from-file .....        | 43 |
| with-input-from-string .....      | 44 |
| with-output-to-file .....         | 43 |
| with-output-to-string .....       | 44 |
| write .....                       | 47 |
| write* .....                      | 47 |
| write-char .....                  | 48 |
| write-with-shared-structure ..... | 47 |

**Z**

|             |    |
|-------------|----|
| zero? ..... | 20 |
|-------------|----|