STklos Reference Manual

(version 1.70)

Erick Gallesio

Table of Contents

Preface

This document provides a complete list of procedures and special forms implemented in version **0.3** of ***STklos***. The language implemented by ***STklos*** is nearly compliant with the language described in the Revised^5 Report on the Algorithmic Language Scheme (aka R^{5}RS) [R5RS].

However, since version 1.30, the goal is to make it more compliant with R^{7}RS [R7RS]. This document states the compliance of each construction relatively to these reports, or if it is a ***STklos*** extension.

Licences

- ***STklos*** program and supporting files are published under the terms of the *GNU General Public Licence version 2 or later*. This licence is available at the following URL: https://www.gnu.org/licenses/gpl-3.0.html
- The manual you're now reading is published under the terms of the *GNU Free Documentation License or later* (see Appendix B).

Chapter 1. Introduction

1.1. Overview of *STklos*

STklos is the successor of ***STk*** [STk], a Scheme interpreter which was tightly connected to the Tk graphical toolkit [Tk]. ***STk*** had an object layer which was called ***STklos***. At this time, ***STk*** was used to denote the base Scheme interpreter and ***STklos*** was used to denote its object layer, which was an extension. For instance, when programming a GUI application, a user could access the widgets at the (low) Tk level, or access them using a neat hierarchy of classes wrapped in ***STklos***.

Since the object layer is now more closely integrated with the language, the new system has been renamed ***STklos*** and ***STk*** is now 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 need heavier work are programs which use Tk without objects, since the new preferred GUI system is now based on GTK+ [GTK]. Programmers used to GUI programming using ***STklos*** classes will find that both system, even if not identical in every points, share the same philosophy.

1.2. Lexical Conventions

1.2.1. Identifiers

In ***STklos***, by default, identifiers which start (or end) with a colon “:” are considered as keywords. For instance `:foo` and `bar:` are ***STklos*** keywords, but `not:key` is not a keyword. Alternatively, to be compatible with other Scheme implementations, the notation `#:foo` is also available to denote the keyword of name `foo`. See *Section 4.14* for more information.

The reader behavior, concerning keywords, can also be changed by the following directives:

- the `#!keyword-colon-position-none` tells the reader that colon in a symbol should not be interpreted as a keyword;
- the `#!keyword-colon-position-before` tells the reader that a symbol starting with a colon should be interpreted as a keyword;
- the `#!keyword-colon-position-after` tells the reader that a symbol ending with a colon should be interpreted as a keyword;
- the `#!keyword-colon-position-both` tells the reader that a symbol starting or ending with a colon should be interpreted as a keyword

In any case, the notation `#:key` is always read as a keyword.

By default, ***STklos*** is case sensitive as specified by R7RS. This behavior can be changed by

- the `--case-insensitive` option used for the commands `stklos` or `stklos-compile`

- the #!fold-case directive which can appear anywhere comments are permitted but must be followed by a delimiter. This directive (and the #!no-fold-case) are treated as comments, except that they affect the reading of subsequent data from the same port. The #!fold-case directive causes subsequent identifiers and character names to be case-folded as if by string-foldcase. It has no effect on character literals. The #!no-fold-case directive causes the reader to a non-folding behavior.

1.2.2. Comments

There are four 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 R^{5}RS).
- multi-lines comment use the classical Lisp convention: a comment begins with #| and ends with |#. This form of comment is now defined by **SRFI-30** (*Nested Multi-line Comments*).
- a sharp sign followed by a semicolon #; comments the next Scheme expression. This is useful to comment a piece of code which spans multiple lines
- 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 executes the script as a Scheme program. This form is compatible with the notation introduced in **SRFI-22** (*Running Scheme Scripts on Unix*).

Note that, as a special case, that the sequences #!key, #!optional and #!rest are respectively converted to the ***STklos*** keywords #:key, #:optional and #:rest. This permits to Scheme lambdas, which accepts keywords and optional arguments, to be compatible with DSSSL lambdas [DSSSL].

1.2.3. Here Strings

Here strings permit to easily enter multilines strings in programs. The sequence of characters #<< starts a here string. The characters following this sequence #<< until a newline character define a terminator for the here string. The content of the string includes all characters between the #<< line and a line whose only content is the specified terminator. No escape sequences are recognized between the starting and terminating lines.

Example: the sequence

```
#<<EOF
abc
def
  ghi
EOF
```

is equivalent to the string

```
"abc\ndef\n  ghi"
```

1.2.4. Other Notations

STk accepts all the notations defined in R^{5}RS plus

- `#true` and `#false` are other names for the constants `#t` and `#f` as proposed by R^{7}RS.
- `[]` 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 (see *Section 4.4*).
- `:` a colon at the beginning or the end of a symbol introduces a keyword. Keywords are described in section Keywords (see _Section 4.14).
- `#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 (cons 'x a)))
  (list a b))
```

can also be represented in this way:

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

Chapter 2. Expressions

This chapter describes the main forms available in ***STklos***. R^{5}RS constructions are given very succinctly here for reference. See [R5RS] for a complete description.

2.1. Literal expressions

R^{5}RS syntax

```
(quote <datum>)
'<datum>
```

The quoting mechanism is identical to R^{5}RS, 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
```

R^{5}RS requires to quote constant lists and constant vectors. This is not necessary with STklos.

2.2. Procedures

STklos syntax

```
(lambda <formals> <body>)
```

A lambda expression evaluates to a procedure. STklos lambda expression have been extended to allow a 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^{5}RS.

`<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^{5}RS.

`(<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^{5}RS.

`(<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 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 collects 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)
```

STklos procedure

```
(closure? obj)
```

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

STklos syntax

```
(case-lambda <clause> …)
```

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 R^{5}RS.

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 … vk`, 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 … vk` 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
```

2.3. Assignments

R^{5}RS syntax

```
(set! <variable> <expression>)
(set! (<proc> <arg> …) <expression>)
```

The first form of set! is the R^{5}RS one:

<Expression> is evaluated, and the resulting value is stored in the location to which <variable> is bound. <Variable> must be bound either in some region enclosing the set! expression or at top level.

```
(define x 2)
(+ x 1)                     =>  3
(set! x 4)                  =>  unspecified
(+ x 1)                     =>  5
```

The second form of set! is defined in **SRFI-17** (*Generalized set!*):

This special form set! is extended so the first operand can be a procedure application, and not just a variable. The procedure is typically one that extracts a component from some data structure. Informally, when the procedure is called in the first operand of `set!`, it causes the corresponding component to be replaced by the second operand. For example,

```
(set (vector-ref x i) v)
```

would be equivalent to:

```
(vector-set! x i v)
```

Each procedure that may be used as the first operand to `set!` must have a corresponding ***setter*** procedure. The procedure `setter` (see below) takes a procedure and returns the corresponding setter procedure. So,

```
(set! (proc arg ...) value)
```

is equivalent to the call

```
((setter proc) arg ... value)
```

The result of the `set!` expression is unspecified.

STklos procedure

```
(setter proc)
```

Returns the setter associated to a `proc`. Setters are defined in the **SRFI-17** (*Generalized set!*) document. A setter proc, can be used in a generalized assignment, as described in `set!`.

To associate `s` to the procedure `p`, use the following form:

```
(set! (setter p) s)
```

For instance, we can write

```
(set! (setter car) set-car!)
```

The following standard procedures have pre-defined setters:

```
(set! (car x) v)              == (set-car! x v)
```

```
(set! (cdr x) v)              == (set-cdr! x v)
(set! (string-ref x i) v)     == (string-set! x i v)
(set! (vector-ref x i) v)     == (vector-set! x i v)!
(set! (slot-ref x 'name) v)   == (slot-set! x 'name v)
(set! (struct-ref x 'name) v) == (struct-set! x 'name v)
```

Furhermore, Parameter Objects (see *Section 4.21*) are their own setter:

```
(real-precision)              => 15
(set! (real-precision) 12)
(real-precision)              => 12
```

2.4. Conditionals

R^{5}RS syntax

```
(if <test> <consequent> <alternate>)
(if <test> <consequent>)
```

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
```

R^{5}RS syntax

```
(cond <clause1> <clause2> ⋯)
```

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
```

R^{7}RS syntax

```
(case <key> <clause1> <clause2> …)
```

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***.

If the selected <clause> or else clause uses the ⇒ alternate form, then the expression is evaluated. It is an error if its value is not a procedure accepting one argument. This procedure is then called on the value of the hkeyi and the values returned by this procedure are returned by the case expression.

```
  (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
  (case (car '(c d))
    ((a e i o u) 'vowel)
    ((w y) 'semivowel)
    (else  => (lambda (x) (x))))  =>  c
```

R^{5}RS syntax

```
(and <test₁> ⋯)
```

The $<test_i>$ 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
```

R^5RS syntax

```
(or <test₁> …)
```

The `<testᵢ>` 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)
```

STklos syntax

```
(when <test> <expression1> <expression2> …)
```

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***.

STklos syntax

```
(unless <test> <expression1> <expression2> …)
```

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***.

2.5. 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.

STklos also provides a `fluid-let` form which is described below.

```
(let <bindings> <body>)
(let <variable> <bindings> <body>)
```

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

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

where each `<init`$_i$`>` 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` 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))
```

R^{5}RS syntax

```
(let <bindings> <body>)*
```

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
```

R^{5}RS syntax

```
(letrec <bindings> <body>)
```

<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
```

R7RS syntax

```
(letrec <bindings> <body>)*
```

<bindings> should have the form as in `let` and body is a sequence of zero or more definitions followed by one or more expressions.

The `<variable>`s are bound to fresh locations, each `variable` is assigned in left-to-right order to the result of evaluating the corresponding `init`, the `body` is evaluated in the resulting environment, and the values of the last expression in `body` are returned. Despite the left-to-right evaluation and assignment order, each binding of a `variable` has the entire `letrec*` expression as its region, making it possible to define mutually recursive procedures. If it is not possible to evaluate each `init` without assigning or referring to the value of the corresponding `variable` or the `variable` of any of the bindings that follow it in `bindings`, it is an error.

```
(letrec* ((p (lambda (x)
               (+ 1 (q (- x 1)))))
          (q(lambda (y)
              (if (zero? y)
                  0
                  (+ 1 (p (- y 1))))))
          (x (p 5))
          (y x))
  y)  => 5
```

R7RS syntax

```
(let-values ((<formals> <expression>) …) <body>)
```

Each `<formals>` should be a formal arguments list as for a `lambda` expression.

The `<expression>`s are evaluated in the current environment, the variables of the `<formals>` are bound to fresh locations, the return values of the `<expression>`s are stored in the variables, the `<body>` is evaluated in the extended environment, and the values of the last expression of `<body>` are returned.

The matching of each `<formals>` to values is as for the matching of `<formals>` to arguments in a `lambda` expression, and it is an error for an `<expression>` to return a number of values that does not match its corresponding `<formals>`.

```
(let-values (((root rem) (exact-integer-sqrt 32)))
   (* root rem))            =>  35

(let ((a 'a) (b 'b) (x 'x) (y 'y))
   (let-values (((a b) (values x y))
                ((x y) (values a b)))
     (list a b x y)))      => (x y a b)
```

R7RS syntax

```
(let-values ((<formals> <expression>) …) <body>)
```

Each `<formals>` should be a formal arguments list as for a `lambda` expression.

`let*-values` is similar to `let-values`, but the bindings are performed sequentially from left to right, and the region of a binding indicated by `(<formals> <expression>)` is that part of the `let*-values` 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 ((a 'a) (b 'b) (x 'x) (y 'y))
   (let*-values (((a b) (values x y))
                ((x y) (values a b)))
     (list a b x y)))      => (x y x y)
```

R7RS procedure

```
(define-values formals expression)
```

The form `define-values` creates multiple definitions from a single expression returning multiple values. Here, `expression` is evaluated, and the `formals` are bound to the return values in the same way that the `formals` in a lambda expression are matched to the arguments in a procedure call.

```
(let ()
  (define-values (x y) (exact-integer-sqrt 17))
  (list x y))                     => (4 1)

(let ()
   (define-values (x y) (values 1 2))
   (+ x y))                       => 3

(let ()
   (define-values (x . y) (values 1 2 3))
```

```
    (list x y)                    => (1 (2 3))
```

STklos syntax

```
(fluid-let <bindings> <body>)
```

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) l)) =>  (out in out in out)
```

2.6. Sequencing

R^{5}RS syntax

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

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
```

STklos syntax

```
(tagbody <expression1> <expression2> ...)
(→ tag)
```

The <expression>s are evaluated sequentially from left to right, and the value(s) of the last <expression> is(are) returned as in a begin form. Within a tagbody form expressions which are keywords are considered as tags and the special form (→ tag) is used to transfer execution to the given tag. This is a **very low level"** form which is inspired on tabgody Common Lisp's form. It can be useful for defining new syntaxes, and should probably not used as is.

```
(tagbody              ;; an infinite loop
   #:1 (display ".")
       (-> #:1))

(let ((v 0))
  (tagbody
   #:top (when (< v 5)
           (display v)
           (set! v (fx+ v 1))
           (-> #:top))))                 |- 01234

(tagbody (display 1)
         (tagbody (display 2)
                  (-> #:inner)
                  (display "not printed")
           #:inner
                 (display 3)
                 (-> #:outer)
                 (display "not printed too"))
   #:outer
         (display "4"))                  |- 1234
```

2.7. Iterations

R^{5}RS syntax

```
(do [[<var1> <init1> <step1>] …] [<test> <expr> …] <command> …)
```

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
```

STklos syntax

```
(dotimes [var count] <expression1> <expression2> …)
(dotimes [var count result] <expression1> <expression2> …)
```

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 l)
     (set! l (cons i l)))) => (3 2 1 0)
```

STklos syntax

```
(repeat count <expression1> <expression2> …)
```

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). The result of `repeat` is undefined.

This form could be easily simulated with `dotimes`. Its interest is that it is faster.

```
(repeat 3 (display "."))     => prints "..."
(repeat 0 (display "."))     => prints nothing
```

STklos syntax

```
(while <test> <expression1> <expression2> …)
```

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

STklos syntax

```
(until <test> <expression1> <expression2> …)
```

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

2.8. Delayed Evaluation

R^{5}RS syntax

```
(delay <expression>)
```

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 for a more complete description of delay.

R^{7}RS syntax

```
(delay-force <expression>)
(lazy <expression>)
```

The expression (delay-force expression) is conceptually similar to (delay (force expression)), with the difference that forcing the result of delay-force will in effect result in a tail call to (force expression), while forcing the result of (delay (force expression)) might not. Thus iterative lazy algorithms that might result in a long series of chains of delay and force can be rewritten using delay-force to prevent consuming unbounded space during evaluation.

The special form delay-force appears with name lazy in **SRFI-45** (*Primitives for Expressing Iterative Lazy Algorithms*).

R^{5}RS procedure

```
(force promise)
```

Forces the value of promise (see *primitive delay*). 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
```

See R5RS for details on a posssible way to implement `force` and `delay`.

R7RS procedure

```
(promise? obj)
```

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

R7RS procedure

```
(make-promise obj)
(eager obj)
```

The `make-promise` procedure returns a promise which, when forced, will return `obj` . It is similar to `delay`, but does not delay its argument: it is a procedure rather than syntax. If `obj` is already a promise, it is returned.

The primitve `make-promise` appears with name `eager` in SRFI-45.

2.9. Quasiquotation

R^{5}RS syntax

```
(quasiquote <template>)
`<template>
```

"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 quasiquotation, and decreases by one inside each unquotation.

```
`(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>)`.

2.10. 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 called ***Macro by Example*** (Eugene Kohlbecker, R4RS) done by Dorai Sitaram. This implementation generates low level STklos macros. This implementation of hygienic macros is not expensive.

The major drawback of this implementation is that the macros are not ***referentially transparent*** (see section **Macros** in R4RS for details). 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 do

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

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

STklos syntax

```
(define-macro (<name> <formals>) <body>)
(define-macro <name> (lambda <formals> <body>))
```

`define-macro` can be used to define low-level macro (i.e. ,(emph "non hygienic") macros). This form is similar to the `defmacro` form of Common Lisp.

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

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

R^{5}RS syntax

```
(define-syntax <identifier> <transformer-spec>)
```

`<Define-syntax>` extends the top-level syntactic environment by binding the `<identifier>` to the specified transformer.

> `<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 ...)))))
```

R^{5}RS syntax

```
(syntax-rules <literals> <syntax-rule> …)
```

`<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.

> For a complete description of the Scheme pattern language, refer to R^{5}RS.

R^{5}RS syntax

```
(let-syntax <bindings> <body>)
```

<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.

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
```

R^{5}RS syntax

```
(letrec-syntax <bindings> <body>)
```

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.

`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
```

STklos procedure

```
(macro-expand form)
(macro-expand form)*
```

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

```
(define-macro (add1 x) `(+ ,x 1))
(macro-expand '(add1 foo)) => (+ foo 1)
(macro-expand '(car bar))  => (car bar)
```

`macro-expand` returns the **full** macro expansion of `form`, that is it repeats the macro-expansion, while the expanded form contains macro calls.

```
(define-macro (add2 x) `(add1 (add1 ,x)))
(macro-expand '(add2 foo)) => (add1 (add1 foo))
(macro-expand* '(add2 foo)) => (+ (+ foo 1) 1)
```

`macro-expand` and `macro-expand*` expand only the global macros.

[1] Documentation about hygienic macros has been stolen in the SLIB manual

Chapter 3. Program structure

R^{5}RS discusses how to structure programs. Everything which is defined in Section 5 of R^{5}RS applies also to ***STklos***. To make things shorter, this aspects will not be described here (see R^{5}RS 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 and Dybvig paper [TuD96]. As their modules system, ***STklos*** modules are defined to be easily used in an interactive environment.

STklos syntax

```
(define-module <name> <expr1> <expr2> ...)
```

`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>`[1]

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 bindings** of a R^{5}RS 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***.

STklos procedure

```
(current-module)
```

Returns the current module.

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

STklos procedure

```
(find-module name)
(find-module name default)
```

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`.

STklos procedure

```
(module? object)
```

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

```
(module? (find-module 'STklos))   => #t
(module? 'STklos)                 => #f
(module? 123 'no)                 => no
```

STklos syntax

```
(export <symbol1> <symbol2> …)
```

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***.

STklos syntax

```
(import <module1> <module2> …)
```

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 last 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 d)
  (define b 'M2-b)
  (define c 'M2-c)
  (define d 'M2-d))

(define-module M3
  (import M1 M2)
  (display (list a b c d)))  |- (M1-a M2-b M2-c M2-d)

(define-module M4
  (import M2 M1)
  (display (list a b c d)))  |- (M1-a M1-b M2-c M2-d)
```

It is also possible to import partially (i.e. not all exported symbols) from a module, as shown below:

```
(define-module M5
  (import (M2 c d) M1)
  (display (list a b c d)))  |- (M1-a M1-b M2-c M2-d)
```

In this case, only the symbols `c` and `d` are imported from module `M2`.

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

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.

STklos syntax

```
(select-module <name>)
```

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`. [2]):

```
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>
```

STklos procedure

```
(symbol-value symbol module)
(symbol-value symbol module default)
```

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`.

STklos procedure

```
(symbol-value* symbol module)
```

```
(symbol-value* symbol module default)
```

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** in the STklos module if module is not a R^{7}RS library.

STklos procedure

```
(module-name module)
```

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

STklos procedure

```
(module-imports module)
```

Returns the list of modules that `module` (fully) imports.

STklos procedure

```
(module-exports module)
```

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.

STklos procedure

```
(module-symbols module)
```

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

STklos procedure

```
(module-symbols* module)
```

Returns the the list of symbols acessible in `module` (that is the symbols defined in `module` and the one defined in the `STklos` module if module is not a R^{7}RS library.

STklos procedure

```
(all-modules)
```

Returns the list of all the living modules.

STklos syntax

```
(in-module mod s)
(in-module mod s default)
```

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,

```
(in-module my-module foo)
```

is equivalent to

```
(symbol-value* 'foo (find-module 'my-module))
```

[1] 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.

[2] This transcript uses the default toplevel loop which displays the name of the current module in the evaluator prompt.

Chapter 4. Standard Procedures

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

R^{5}RS procedure

```
(eqv? obj1 obj2)
```

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
  ```

  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*, and are either both exact or both inexact.
- `obj1` and `obj2` are both characters and are the same character according to the `char=?` *procedure*`.
- 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.

STklos extends R^{5}RS 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 :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
```

In fact, the value returned by STklos depends on the way code is entered and can yield #t in some cases and #f in others.

See R^{5}RS for more details on eqv?.

R^{5}RS procedure

```
(eq? obj1 obj2)
```

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 that:

- STklos extends R^{5}RS `eq?` to take into account the keyword type.
- 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
```

R^{5}RS procedure

```
(equal? obj1 obj2)
```

`Equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers and symbols. A rule of thumb is that objects are generally `equal?` if they print the same. `Equal?` always terminates even 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
(equal? '#1=(a b . #1#)
        '#2=(a b a b . #2#))    =>  #t
```

A rule of thumb is that objects are generally `equal?` if they print the same.

4.2. Numbers

R^{5}RS description of numbers is quite long and will not be given here. STklos support the full number tower as described in R^{5}RS; see this document for a complete description.

STklos extends the number syntax of R5RS with the following inexact numerical constants: `+inf.0` (infinity), `-inf.0` (negative infinity), `+nan.0` (not a number), and `-nan.0` (not a number).

R^{5}RS procedure

```
(number? obj)
(complex? obj)
(real? obj)
(rational? obj)
(integer? obj)
```

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 for a number then all higher type predicates are also true for 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 `(and (finite? x) (= x (round x)))`

```
(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? 3.2)          =>  #f
(integer? 8/4)          =>  #t
(integer? "no")         =>  #f
(complex? +inf.0)       =>  #t
(real? -inf.0)          =>  #t
(rational? +inf.0)      =>  #f
(integer? -inf.0)       =>  #f
```

R5RS procedure

```
(exact? z)
(inexact? z)
```

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

R7RS procedure

```
(inexact z)
(exact z)
```

These R7RS procedures correspond to the R5RS `exact→inexact` and `inexact→exact` procedure respectively

R7RS procedure

```
(exact→integer? z)
```

Returns #t if z is both exact and an integer; otherwise returns #f.

```
(exact-integer? 32)   => #t
(exact-integer? 32.0) => #f
(exact-integer? 32/5) => #f
```

STklos procedure

```
(bignum? x)
```

This predicates returns `#t` if `x` is an integer number too large to be represented with a native integer.

```
(bignum? (expt 2 300))     => `#t`    ;; (very likely)
(bignum? 12)               => `#f`
(bignum? "no")             => `#f`
```

R5RS procedure

```
(= z1 z2 z3 …)
(< x1 x2 x3 …)
(> x1 x2 x3 …)
(⇐ x1 x2 x3 …)
(>= x1 x2 x3 …)
```

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

```
(= +inf.0 +inf.0)          =>  #t
(= -inf.0 +inf.0)          =>  #f
(= -inf.0 -inf.0)          =>  #t
```

For any finite real number x:

```
(< -inf.0 x +inf.0)        =>  #t
(> +inf.0 x -inf.0)        =>  #t
```

R^{5}RS procedure

```
(finite? z)
(infinite? z)
(zero? z)
(positive? x)
(negative? x)
(odd? n)
(even? n)
```

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

```
(positive? +inf.0)         ==>  #t
(negative? -inf.0)         ==>  #t
(finite? -inf.0)           ==>  #f
(infinite? +inf.0)         ==>  #t
```

R^{7}RS procedure

```
(nan? z)
```

The `nan?` procedure returns #t on `+nan.0`, and on complex numbers if their real or imaginary parts or both are `+nan.0`. Otherwise it returns #f.

```
(nan? +nan.0)          =>  #t
(nan? 32)              =>  #f
(nan? +nan.0+5.0i)     =>  #t
(nan? 1+2i)            =>  #f
```

STklos procedure

```
(make-nan negative? quiet? payload)
(make-nan negative? quiet? payload float)
```

Returns a NaN whose sign bit is equal to `negative?` (`#t` for negative, `#f` for positive), whose quiet bit is equal to quiet? (`#t` for quiet, `#f` for signaling), and whose payload is the positive exact integer payload. It is an error if payload is larger than a NaN can hold.

The optional parameter `float`, is never used in ***STklos***.

This function is defined in **SRFI-208**.

STklos procedure

```
(nan-negative? nan)
```

returns `#t` if the sign bit of `nan` is set and `#f` otherwise.

STklos procedure

```
(nan-quiet? nan)
```

returns `#t` if `nan` is a quiet NaN.

STklos procedure

```
(nan-payload nan)
```

returns the payload bits of `nan` as a positive exact integer.

STklos procedure

```
(nan=? nan1 nan2)
```

Returns #t if nan1 and nan2 have the same sign, quiet bit, and payload; and #f otherwise.

R^{5}RS procedure

```
(max x1 x2 …)
(min x1 x2 …)
```

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)              =>  4    ; exact
(max 3.9 4)            =>  4.0  ; inexact
```

For any real number x:

```
(max +inf.0 x)         =>  +inf.0
(min -inf.0 x)         =>  -inf.0
```

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

R^{7}RS procedure

```
(floor/ n1 n2)
(floor-quotient n1 n2)
(floor-remainder n1 n2)
(truncate/ n1 n2)
(truncate-quotient n1 n2)
(truncate-remainder n1 n2)
```

These procedures implement number-theoretic (integer) division. It is an error if n2 is zero. The procedures ending in '/' return two integers; the other procedures return an integer. All the procedures compute a quotient q and remainder r such that n1=n2*q+r.

See R^{7}RS for more information.

```
(floor/ 5 2)         => 2 1
(floor/ -5 2)        => -3 1
(floor/ 5 -2)        => -3 -1
(floor/ -5 -2)       => 2 -1
(truncate/ 5 2)      => 2 1
```

```
(truncate/ -5 2)      => -2 -1
(truncate/ 5 -2)      => -2 1
(truncate/ -5 -2)     => 2 -1
(truncate/ -5.0 -2)  => 2.0 -1.0%
```

R^{5}RS procedure

```
(+ z1 …)
```
(z1 ...)*

These procedures return the sum or product of their arguments.

```
(+ 3 4)                  =>  7
(+ 3)                    =>  3
(+)                      =>  0
(+ +inf.0 +inf.0)        =>  +inf.0
(+ +inf.0 -inf.0)        =>  +nan.0
(* 4)                    =>  4
(*)                      =>  1
(* 5 +inf.0)             =>  +inf.0
(* -5 +inf.0)            =>  -inf.0
(* +inf.0 +inf.0)        =>  +inf.0
(* +inf.0 -inf.0)        =>  -inf.0
(* 0 +inf.0)             =>  +nan.0
```

For any finite number z:

```
      (+ +inf.0 z)      =>  +inf.0
      (+ -inf.0 z)      =>  -inf.0
```

R^{5}RS procedure

```
(- z)
(- z1 z2)
(/ z)
(/ z1 z2 …)
```

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.

```
(- 3 4)                  =>  -1
(- 3 4 5)                 =>  -6
(- 3)                     =>  -3
(- +inf.0 +inf.0)         => +nan.0
(/ 3 4 5)                 =>  3/20
(/ 3)                     =>  1/3
(/ 0.0)                   => +inf.0
(/ 0)                     => error (division by 0)
```

R^5RS procedure

```
(abs x)
```

`Abs` returns the absolute value of its argument.

```
(abs -7)                  =>  7
(abs -inf.0)              => +inf.0
```

R^5RS procedure

```
(quotient n1 n2)
(remainder n1 n2)
(modulo n1 n2)
```

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 < abs(nr) < abs(n2), 0 < abs(nm) < 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
```

R^{5}RS procedure

```
(gcd n1 …)
(lcm n1 …)
```

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
```

R^{5}RS procedure

```
(numerator q)
(denominator q)
```

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
```

R^{5}RS procedure

```
(floor x)
(ceiling x)
(truncate x)
(round x)
```

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.

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

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
```

R^{5}RS procedure

```
(rationalize x y)
```

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
```

R^{5}RS procedure

```
(exp z)
(log z)
(log z b)
(sin z)
(cos z)
(tan z)
(asin z)
(acos z)
(atan z)
(atan y x)
```

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 `log` computes the logarithm of x in base b as

```
(/ (log x) (log b))
```

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.

STklos procedure

```
(sinh z)
(cosh z)
(tanh z)
(asinh z)
(acosh z)
```

```
(atanh z)
```

These procedures compute the hyperbolic trigonometric functions.

```
(sinh 1)     => 1.1752011936438
(sinh 0+1i)  => 0.0+0.841470984807897i
(cosh 1)     => 1.54308063481524
(cosh 0+1i)  => 0.54030230586814
(tanh 1)     => 0.761594155955765
(tanh 0+1i)  => 0.0+1.5574077246549i
(asinh 1)    => 0.881373587019543
(asinh 0+1i) => 0+1.5707963267949i
(acosh 0)    => 0+1.5707963267949i
(acosh 0+1i) => 1.23340311751122+1.5707963267949i
(atanh 1)    => error
(atanh 0+1i) => 0.0+0.785398163397448i
```

In general, `(asinh (sinh x))` and similar compositions should be equal to `x`, except for inexactness due to the internal floating point number approximation for real numbers.

```
(sinh (asinh 0+1i)) => 0.0+1.0i
(cosh (acosh 0+1i)) => 8.65956056235493e-17+1.0i
(tanh (atanh 0+1i)) => 0.0+1.0i
```

These functions will always return an exact result for the following arguments:

```
(sinh 0.0)     => 0
(cosh 0.0)     => 1
(tanh 0.0)     => 0
(asinh 0.0)    => 0
(acosh 1.0)    => 0
(atanh 0.0)    => 0
```

R^{5}RS procedure

```
(sqrt z)
```

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

R^{7}RS procedure

```
(square z)
```

Returns the square of `z`. This is equivalent to `(* z z)`.

```
(square 42)      => 1764
(square 2.0)     => 4.0
```

R7RS procedure

```
(exact-integer-sqrt k)
```

Returns two non negatives integers `s` and `r` where `k=s2+r and k<(s+1)2`.

```
(exact-integer-sqrt 4)      => 2 0
(exact-integer-sqrt 5)      => 2 1
```

R5RS procedure

```
(expt z1 z2)
```

Returns `z1` raised to the power `z2`.

`0,(sup "z")` is 1 if `z = 0` and `0` otherwise.

R5RS procedure

```
(make-rectangular x1 x2)
(make-polar x3 x)
(real-part z)
(imag-part z)
(magnitude z)
(angle z)
```

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

`z = x1 + x2.i = x3 . e,(sup "i.x4")`

Then

```
(make-rectangular x1 x2)       => z
(make-polar x3 x4)             => z
(real-part z)                  => x1
(imag-part z)                  => x2
(magnitude z)                  => abs(x3)
(angle z)                      => xa
```

where `-,(symbol "pi") < xa ⇐ ,(symbol "pi")` with `xa = x4 + 2,(symbol "pi")n` for some integer n.

```
(angle +inf.0)                 => 0.0
(angle -inf.0)                 => 3.14159265358979
```

`Magnitude` is the same as `abs` for a real argument.

R5RS procedure

```
(exact→inexact z)
(inexact→exact z)
```

`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.

R5RS procedure

```
(number→string z)
(number→string z radix)
```

`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.

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.

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.

R^{5}RS procedure

```
(string→number string)
(string→number string radix)
```

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. ,(code "\"#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
(string->number "+inf.0")     =>  +inf.0
(string->number "-inf.0")     =>  -inf.0
```

STklos procedure

```
(bit-and n1 n2 …)
(bit-or n1 n2 …)
(bit-xor n1 n2 …)
(bit-not n)
(bit-shift n m)
```

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 ,(emph "and"), inclusive and exclusive ,(emph "or"). `bit-not` returns the bitwise ,(emph "not") of `n`. `bit-shift` returns the bitwise ,(emph "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
```

STklos procedure

```
(random-integer n)
```

Return an integer in the range \[0, ..., n\[. Subsequent results of this procedure appear to be independent uniformly distributed over the range \[0, ..., 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 ,(link-srfi 27) definition for more details).

STklos procedure

```
(random-real)
```

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 ,(link-srfi 27) definition for more details).

STklos procedure

```
(decode-float n)
```

decode-float returns three exact integers: significand, exponent and sign (where -1 ⇐ sign ⇐ 1). The values returned by decode-float satisfy:

```
n = (* sign significand (expt 2 exponent))
```

Here is an example of decode-float usage.

```
(receive l (decode-float -1.234) l)
                  => (5557441940175192 -52 -1)
(exact->inexact (* -1
                  5557441940175192
                  (expt 2 -52)))
                  => -1.234
```

STklos procedure

```
(integer-length n)
```

`Integer-length` returns the necessary number of bits to represent `n` in 2's complement, assuming a leading 1 bit when `n` is negative. When `n` is zero, the procedure returns zero. This procedure works for any type of integer (fixnums and bignums).

```
(integer-length -3)            => 2
(integer-length -2)            => 1
(integer-length -1)            => 0
(integer-length 0)             => 0
(integer-length 1)             => 1
(integer-length 2)             => 2
(integer-length 3)             => 2
(integer-length (expt 2 5000)) => 5001
```

4.2.1. Fixnums

STklos defines small integers as fixnums. Operations on fixnums are generally faster than operations which accept general numbers. Fixnums operations, as described below, may produce results which are incorrect if some temporary computation falls outside the range of fixnum. These functions should be used only when speed really matters.

The functions defined in this section are conform to the ones defined in **SRFI-143** (*Fixnums*)

STklos procedure

```
(fixnum? obj)
```

Returns `#t` if obj is an exact integer within the fixnum range, `#f` otherwise.

STklos procedure

```
(fixnum-width)
```

Returns the number of bits used to represent a fixnum number.

STklos procedure

```
(least-fixnum)
(greatest-fixnum)
```

These procedures return the minimum value and the maximum value of the fixnum range.

STklos procedure

```
(fxzero? obj)
```

`fxzero?` returns `#t` if `obj` is the fixnum zero and returns `#f` if it is a non-zero fixnum.

```
(fxzero? #f)              =>  error
(fxzero? (expt 100 100)) =>  error
(fxzero? 0)               =>  #t
(fxzero? 1)               =>  #f
```

STklos procedure

```
(fxpositive? obj)
(fxnegative? obj)
```

`fxpositive?` returns `#t` if `obj` is a positive fixnum and returns `#f` if it is a non-positive fixnum. `fxnegative?` can be used to test if a fixnum is negative.

```
(fxpositive? #f)              =>  error
(fxpositive? (expt 100 100)) =>  error
(fxpositive? 0)               =>  #f
(fxpositive? 1)               =>  #t
(fxpositive? -1)              =>  #f
(fxnegative? 0)               =>  #f
(fxnegative? 1)               =>  #f
(fxnegative? -1)              =>  #t
```

STklos procedure

```
(fxodd? obj)
```

fxodd? returns #t if obj is a odd fixnum and returns #f if it is an even fixnum.

```
(fxodd? #f)              =>  error
(fxodd? (expt 100 100)) =>  error
(fxodd? 0)               =>  #f
(fxodd? 1)               =>  #t
(fxodd? 4)               =>  #f
(fxeven? 0)              =>  #t
(fxeven? 1)              =>  #f
(fxeven? 4)              =>  #t
```

STklos procedure

```
(fx+ fx1 fx2)
(fx- fx1 fx2)
(fx fx1 fx2)*
(fxquotient fx1 fx2)
(fxremainder fx1 fx2)
(fxmodulo fx1 fx2)
(fxabs fx)
(fxneg fx)
```

These procedures compute (respectively) the sum, the difference, the product, the quotient and the remainder and modulo of the fixnums fx1 and fx2. The call of fx- with one parameter fx computes the opposite of fx, and is equivalent in a call of fxneg with this parameter. fxabs computes the absolute value of fx.

STklos procedure

```
(fxsquare fx1)
(fxsqrt fx1)
```

These procedures compute (respectively) the square and the square root of the fixnum fx1. fxsqrt id semantically equivalent to exact-integer-sqrt (not sqrt), so that (fxsqrt n) returns two values a, b, such that a*a+b=n.

```
(fxsqrt #f)              =>  error
(fxsqrt (expt 100 100)) =>  error
(fxsqrt -1)              =>  error
(fxsqrt 0)               =>  0, 0
(fxsqrt 1)               =>  1, 0
```

```
  (fxsqrt 6)              =>  2, 2
```

STklos procedure

```
(fxmax fx1 fx2 ...)
(fxmin fx1 fx2 ...)
```

These procedures return the maximum or minimum of their fixnum arguments.

```
(fxmax 3 4)              =>  4
(fxmax 3.9 4)            =>  error
(fxmax)                  =>  error
(fxmax 2 -1 3)           =>  3
```

STklos procedure

```
(fx<? fx1 fx2 ...)
(fx<=? fx1 fx2 ...)
(fx>? fx1 fx2 ...)
(fx>=? fx1 fx2 ...)
(fx=? fx1 fx2 ...)
```

These are SRFI-143 procedures that compare the fixnums `fx1`, `fx2`, and so on. `fx<?` and `fx>?` return `#t` if the arguments are in strictly increasing/decreasing order; `fx<=?` and `fx>=?` do the same, but admit equal neighbors; `fx=?` returns `#t` if the arguments are all equal.

STklos procedure

```
(fxnot fx1)
(fxand fx ...)
(fxior fx ...)
(fxxor fx ...)
```

These procedures are specified in SRFI-143, and they return (respectively) the bitwise not, and, inclusive or and exclusive or of their arguments, which must be fixnums.

```
(fxnot 1)              => -2
(fxnot 0)              => -1
(fxand #x1010 #x1011)  => 4112  ; = #x1010
(fxior #x1010 #x1011)  => 4113  ; = #x1011
```

```
(fxxor #x1010 #x1011)  => 1      ; = #x0001
```

STklos procedure

```
(fxarithmetic-shift-right fx count)
(fxarithmetic-shift-left fx count)
(fxarithmetic-shift fx count)
```

These procedures are specified in SRFI-143, and they perform bitwise right-shift, left-shft and shift with arbitrary direction on fixnums. The strictly left and right shifts are more efficient.

```
(fxarithmetic-shift-right #b100110 3) =>   4 ; = #b100
(fxarithmetic-shift-left  #b100110 3) => 304 ; = #b100110000
(fxarithmetic-shift #b101 2)          => 20  ; = #b10100
(fxarithmetic-shift #b101 -2)         =>  1  ; =#b1
```

STklos procedure

```
(fxlength fx)
```

This is a SRFI-143 procedure that returns the length of the fixnum in bits (that is, the number of bits necessary to represent the number).

```
(fxlength #b101)          =>  3
(fxlength #b1101)         =>  4
(fxlength #b0101)         =>  3
```

STklos procedure

```
(fxif mask fx1 fx2)
```

This is a SRFI-143 procedure that merge the fixnum bitstrings `fx1` and `fx2`, with bitstring mask determining from which string to take each bit. That is, if the kth bit of mask is 1, then the kth bit of the result is the kth bit of `fx1`, otherwise the kth bit of `fx2`.

```
(fxif 3 1 8)                            => 9
(fxif 3 8 1)                            => 0
(fxif 1 1 2)                            => 3
```

```
(fxif #b00111100 #b11110000 #b00001111) => #b00110011
```

STklos procedure

```
(fxbit-set? index fx)
```

This is a SRFI-143 procedure that returns `#t` if the `index`-th bit of `fx`.

```
(fxbit-set? 1 3)          => #t
(fxbit-set? 2 7)          => #t
(fxbit-set? 3 6)          => #f
(fxbit-set? 5 #b00111100) => #t
```

STklos procedure

```
(fxcopy-bit index fx value)
```

This is a SRFI-143 procedure that sets the `index`-th bit if `fx` to one if `value` is `#t`, and to zero if `value` is `#f`.

```
(fxcopy-bit 2 3 #t)          =>  7
(fxcopy-bit 2 7 #f)          =>  3
(fxcopy-bit 5 #b00111100 #f) => 28 ; = #b00011100
```

STklos procedure

```
(fxbit-count fx1)
```

This is a SRFI-143 procedure that returns the quantity of bits equal to one in the fixnum `fx` (that is, computes its Hamming weight).

```
(fxbit-count 8)                   => 1
(fxbit-count 3)                   => 2
(fxbit-count 7)                   => 3
(fxbit-count #b00111010)          => 4
```

STklos procedure

```
(fxfirst-set-bit fx1)
```

This is a SRFI-143 procedure that returns the index of the first (smallest index) 1 bit in bitstring `fx`. Returns -1 if `fx` contains no 1 bits (i.e., if `fx` is zero).

```
(fxfirst-set-bit  8)                    => 3
(fxfirst-set-bit  3)                    => 0
(fxfirst-set-bit  7)                    => 0
(fxfirst-set-bit  #b10110000)           => 4
```

STklos procedure

```
(fxbit-field fx1 start end)
```

This is a SRFI-143 procedure that extracts a bit field from the fixnum `fx1`. The bit field is the sequence of bits between `start` (including) and `end` (excluding)

```
(fxbit-field  #b10110000 3 5)  => 6 ; = #b110
```

STklos procedure

```
(fxbit-field-rotate fx)
```

This is a SRFI-143 procedure that returns fx with the field cyclically permuted by count bits towards high-order.

```
(fxbit-field-rotate #b101011100 -2 1 5)     => 342  = #b101010110
(fxbit-field-rotate #b101011011110 -3 2 10) => 3034 = #b101111011010
(fxbit-field-rotate #b101011011110 3 2 10)  => 2806 = #b101011110110
```

STklos procedure

```
(fxbit-field-reverse fx)
```

This is a SRFI-143 procedure that returns fx with the order of the bits in the field reversed.

```
(fxbit-field-reverse #b101011100 1 5)     => #b101001110
(fxbit-field-reverse #b101011011110 2 10) => #b101110110110
```

STklos procedure

```
(fx+/carry i j k)
```

Returns two values: `i`+`j`+`k`, and carry: it is the value of the computation

```
(let*-values (((s) (+ i j k))
              ((q r) (balanced/ s (expt 2 fx-width))))
  (values r q))
```

STklos procedure

```
(fx-/carry i j k)
```

Returns two values: `i`-`j`-`k`, and carry: it is the value of the computation

```
(let*-values (((s) (- i j k))
              ((q r) (balanced/ s (expt 2 fx-width))))
  (values r q))
```

STklos procedure

(fx/**carry i j k)***

Returns two values: `i`*`j`+`k`, and carry: it is the value of the computation

```
(let*-values (((s) (+ (* i j) k))
              ((q r) (balanced/ s (expt 2 fx-width))))
  (values r q))
```

4.3. 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.

R^{5}RS procedure

```
(not obj)
```

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
```

R^{5}RS procedure

```
(boolean? obj)
```

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

```
(boolean? #f)         =>  #t
(boolean? 0)          =>  #f
(boolean? '())        =>  #f
```

R^{5}RS procedure

```
(boolean=? boolean1 boolean2 …)
```

Returns #t if all the arguments are booleans and all are #t or all are #f.

4.4. Pairs and lists

R^{5}RS procedure

```
(pair? obj)
```

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

R^{5}RS procedure

```
(cons obj1 obj2)
```

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)
```

R^{5}RS procedure

```
(car pair)
```

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
```

R^{5}RS procedure

```
(cdr pair)
```

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
```

R5RS procedure

```
(set-car! pair obj)
```

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
```

R5RS procedure

```
(set-cdr! pair obj)
```

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

R5RS procedure

```
(caar pair)
(cadr pair)
...
(cdddar pair)
(cddddr pair)
```

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.

R5RS procedure

```
(null? obj)
```

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

STklos procedure

```
(pair-mutable? obj)
```

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
```

R^{5}RS procedure

```
(list? obj)
```

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
```

R^{7}RS procedure

```
(make-list k)
(make-list k fill)
```

Returns a newly allocated list 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.

R^{5}RS procedure

```
(list obj …)
```

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)            =>  (a 7 c)
(list)                          =>  ()
```

STklos procedure

`(list obj ...)*`

`list*` is like `list` except that the last argument to `list*` is used as the ,(emph "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*)              => ()
```

R^5RS procedure

`(length list)`

Returns the length of `list`.

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

R^5RS procedure

`(append list …)`

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
```

STklos procedure

```
(append! list ...)
```

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 l4))  => ((1 2 3 4 5) (3 4 5) (4 5) (1 2 3 4 5))
```

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

R^{5}RS procedure

```
(reverse list)
```

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)
```

STklos procedure

```
(reverse! list)
```

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
```

R5RS procedure

```
(list-tail list k)
```

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)))))
```

STklos procedure

```
(last-pair list)
```

Returns the last pair of `list`.

```
(last-pair '(1 2 3))   => (3)
(last-pair '(1 2 . 3)) => (2 . 3)
```

R5RS procedure

```
(list-ref list k)
```

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
```

R7RS procedure

```
(list-set! list k obj)
```

The `list-set!` procedure stores `obj` in element `k` of `list`. It is an error if `k` is not a valid index of `list`.

```
(let ((ls (list 'one 'two 'five!)))
   (list-set! ls 2 'three)
   ls)                               => (one two three)
(list-set! '(0 1 2) 1 "oops")        => error (constant list)
```

R^{5}RS / R^{7}RS procedure

```
(memq obj list)
(memv obj list)
(member obj list)
(member obj list compare)
```

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 `compare`, if given, and `equal?` otherwise.

```
   (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)
   (member "B"
           '("a" "b" "c")
           string-ci=?)            => ("b" "c")
   (memv 101 '(100 101 102))       =>  (101 102)
```

As in R^{7}RS, the `member` function accepts also a comparison function.

R^{5}RS / R^{7}RS procedure

```
(assq obj alist)
(assv obj alist)
(assoc obj alist)
(assoc obj alist compare)
```

`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))
(assoc 2.0 '((1 1) (2 4) (3 9)) =)
                                 => (2 4)
(assv 5 '((2 3) (5 7) (11 13)))
                                 =>  (5 7)
```

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.

As in R^{7}RS, the assoc function accepts also a comparison function.

R^{7}RS procedure

```
(list-copy obj)
```

list-copy 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 list-copy on the car and cdr of obj, respectively.

STklos procedure

```
(filter pred list)
(filter! pred list)
```

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 as 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.

STklos procedure

```
(remove pred list)
```

`Remove` returns `list` without the elements 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. `Remove!` does the same job than `remove` by physically modifying its `list` argument

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

STklos procedure

```
(delete x list [=])
(delete! x list [=])
```

`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.

4.5. 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 `|`.

In addition, any character can be used within an identifier when specified via an inline hex escape . For example, the identifier `H\x65;llo` is the same as the identifier Hello, and, if the UTF-8 encoding is used, the identifier `\x3BB;` is the same as the identifier `λ`.

```
'|a|                   => a
(string->symbol "a")  => |A|
(symbol->string '|A|) => "A"
'|a  b|                => |a  b|
'a|B|c                 => |aBc|
(write '|FoO|)         |- |FoO|
(display '|FoO|)       |- FoO
```

R^{5}RS procedure

```
(symbol? obj)
```

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
```

R^{5}RS procedure

```
(symbol=? symbol1 symbol2 …)
```

Returns #t if all the arguments are symbols and all have the same name in the sense of `string=?`.

R^{5}RS procedure

```
(symbol→string string)
```

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"
```

R^{5}RS procedure

```
(string→symbol string)
```

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
```

STklos procedure

```
(string→unterned-symbol string)
```

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)
```

STklos procedure

```
(gensym)
(gensym prefix)
```

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
```

4.6. Characters

The following table gives the list of allowed character names with their ASCII eqivalent expressed in octal. Some chracaters have an alternate name which is also shown in this table.

name	value	alt. name	name	value	alt. name
nul	000	null	soh	001	
stx	002		etx	003	
eot	004		enq	005	
ack	006		bel	007	alarm
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	

name	value	alt. name	name	value	alt. name
rs	036		us	037	
sp	040	space	del	177	delete

STklos supports the complete Unicode character set, if UTF-8 encoding is used. Hereafter, are some examples of characters:

```
#\A                  => uppercase A
#\a                  => lowercase a
#\x41                => the U+0041 character (uppercase A)
#\x03bb              => λ
```

R⁵RS procedure

```
(char? obj)
```

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

R⁵RS procedure

```
(char=? char1 char2 …)
(char<? char1 char2 …)
(char>? char1 char2 …)
(char⇐? char1 char2 …)
(char>=? char1 char2 …)
```

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.)

R⁵RS procedure

```
(char-ci=? char1 char2 …)
(char-ci<? char1 char2 …)
(char-ci>? char1 char2 …)
```

```
(char-ci⇐? char1 char2 …)
(char-ci>=? char1 char2 …)
```

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`.

R^{5}RS procedure

```
(char-alphabetic? char)
(char-numeric? char)
(char-whitespace? char)
(char-upper-case? letter)
(char-lower-case? letter)
```

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.

R^{5}RS procedure

```
(char→integer char)
(integer→char n)
```

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

```
   (char<=? a b) => #t  and  (<= x y) => #t
```

and x and y are in the domain of `integer→char`, then

```
   (<= (char->integer a)
       (char->integer b))         =>  #t

  (char<=? (integer->char x)
           (integer->char y))     =>  #t
```

`integer→char` accepts an exact number between 0 and #xD7FFF or between #xE000 and #x10FFFF, if UTF8 encoding is used. Otherwise, it accepts a number between 0 and #xFF.

R^{5}RS procedure

```
(char-upcase char)
(char-downcase char)
```

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.

STklos procedure

```
(char-foldcase char)
```

This procedure applies the Unicode simple case folding algorithm and returns the result. Note that language-sensitive folding is not used. If the argument is an uppercase letter, the result will be either a lowercase letter or the same as the argument if the lowercase letter does not exist.

R^{7}RS procedure

```
(digit-value char)
```

This procedure returns the numeric value (0 to 9) of its argument if it is a numeric digit (that is, if char-numeric? returns #t), or #f on any other character.

```
(digit-value
(digit-value #\3)        => 3
(digit-value #\x0664)    => 4
(digit-value #\x0AE6)    => 0
(digit-value #\x0EA6)    => #f
```

4.7. Strings

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

Sequence	Character inserted
\a	Alarm
\b	Backspace
\e	Escape

Sequence	Character inserted
\n	Newline
\t	Horizontal Tab
\r	Carriage Return
\"	doublequote U+0022
\\	backslash U+005C
\0abc	ASCII character with octal value abc
\x<hexa value>;	ASCII character with given hexadecimal value
\<intraline whitespace><newline><intraline whitespace>	None (permits to enter a string on several lines)
\<other>	<other>

For instance, the string

```
"ab\040\x20;c\nd\
         e"
```

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

Notes:

- Using octal code is limited to characters in the range 0 to #xFF. It is then not convenient to enter Unicode characters. This form is deprecated should not be used anymore.
- A line ending which is preceded by <intraline whitespace> expands to nothing (along with any trailing <intraline whitespace>), and can be used to indent strings for improved legibility.

R^{5}RS procedure

```
(string? obj)
```

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

R^{5}RS procedure

```
(make-string k)
(make-string k char)
```

`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.

R^{5}RS procedure

```
(string char …)
```

Returns a newly allocated string composed of the arguments.

R^{5}RS procedure

```
(string-length string)
```

Returns the number of characters in the given `string`.

R^{5}RS procedure

```
(string-ref string k)
```

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

R^{5}RS procedure

```
(string-set! string k char)
```

`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
```

R^{5}RS / R^{7}RS procedure

```
(string=? string1 string2 …)
(string-ci=? string1 string2 …)
```

Returns #t if all the 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.

R^{5}RS version of these functions accept only two arguments.

R^{5}RS / R^{7}RS procedure

```
(string<? string1 string2 …)
(string>? string1 string2 …)
(string<=? string1 string2 …)
(string>=? string1 string2 …)
(string-ci<? string1 string2 …)
(string-ci>? string1 string2 …)
(string-ci<=? string1 string2 …)
(string-ci>=? string1 string2)
```

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.

R^{5}RS version of these functions accept only two arguments.

R^{5}RS procedure

```
(substring string start end)
```

String must be a string, and start and end must be exact integers satisfying

```
0 <= start <= end <= (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).

R^{5}RS procedure

```
(string-append string …)
```

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

R^{5}RS / R^{7}RS procedure

```
(string→list string)
(string→list string start)
(string→list string start end)
(list→string list)
```

`String→list` returns a newly allocated list of the characters of `string` between `start` and `end`. `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.

The R^{5}RS version of `string→list` accepts only one parameter.

R^{5}RS / R^{7}RS procedure

```
(string-copy string)
(string-copy string start)
(string-copy string start stop)
```

Returns a newly allocated copy of the part of the given `string` between `start` and `stop`.

The R^{5}RS version of `string-copy` accepts only one argument.

R^{7}RS procedure

```
(string-copy! to at from)
(string-copy! to at from start)
(string-copy! to at from start end)
```

Copies the characters of `string` from between `start` and `end` to string `to`, starting at `at`. The order in which characters are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if `at` is less than zero or greater than the length of `to`. It is also an error if `(- (string-length to) at)` is less than `(- end start)`.

STklos procedure

```
(string-split str)
(string-split str delimiters)
```

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")
```

STklos procedure

```
(string-position str1 str2)
```

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

```
(string-position "ca" "abracadabra") =>  4
(string-position "ba" "abracadabra") =>  #f
```

This function was also called string-index. This name is deprecated since it conficts with the string-index defined in SRFI-13.

STklos procedure

```
(string-find? str1 str2)
```

Returns #t if str1 appears somewhere in str2; otherwise returns #f.

R^{7}RS procedure

```
(string-fill! string char)
(string-fill! string char start)
(string-fill! string char start end)
```

Stores char in every element of the given string between start and end.

The R^{5}RS version of string-fill! accepts only one argument.

STklos procedure

```
(string-blit! s1 s2 offset)
```

This function places the characters of string `s2` in the string `s1` starting at position `offset`. The result of `string-blit!` may modify the string `s1`. Note that the characters of `s2` can be written after the end of `s1` (in which case a new string is allocated).

```
(string-blit! (make-string 6 #\X) "abc" 2)
              => "XXabcX"
(string-blit! (make-string 10 #\X) "abc" 5)
              => "XXXXXabcXX"
(string-blit! (make-string 6 #\X) "a" 10)
              => "XXXXXX\0\0\0\0a"
```

STklos procedure

```
(string-mutable? obj)
```

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
```

The following string primitives are compatible with **SRFI-13** (*String Library*) and their documentation comes from the SRFI document.

Notes:

- The string SRFI is supported by ***STklos***. The function listed below just don't need to load the full SRFI to be used
- The functions `string-upcase`, `string-downcase` and `string-foldcase` are also defined in R^{7}RS.

R^{7}RS procedure

```
(string-downcase str)
(string-downcase str start)
(string-downcase str start end)
```

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 omited, it defaults to 0. If end is omited, 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"
```

In R^{7}RS, string-downcase accepts only one argument.

STklos procedure

```
(string-downcase! str)
(string-downcase! str start)
(string-downcase! str start end)
```

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"
```

R^{7}RS procedure

```
(string-upcase str)
(string-upcase str start)
(string-upcase str start end)
```

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 omited, it defaults to 0. If end is omited, it defaults to the length of str.

In R^{7}RS, string-upcase accepts only one argument.

STklos procedure

```
(string-upcase! str)
(string-upcase! str start)
(string-upcase! str start end)
```

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

STklos procedure

```
(string-titlecase str)
(string-titlecase str start)
(string-titlecase str start end)
```

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 omited, it defaults to 0. If `end` is omited, 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"
```

STklos procedure

```
(string-titlecase! str)
(string-titlecase! str start)
(string-titlecase! str start end)
```

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

STklos procedure

```
(string-append! string …)
```

Extends string by appending each value (in order) to the end of string. A value can be a character or a string.

It is guaranteed that string-append! will return the same object that was passed to it as first argument, whose size may be larger.

This function is defined in SRFI-118.

STklos procedure

```
(string-replace! dst dst-start dst-end src)
(string-replace! dst dst-start dst-end src src-start)
(string-replace! dst dst-start dst-end src src-start src-end)
```

Replaces the characters of the variable-size string dst (between dst-start and dst-end) with the characters of the string src (between src-start and src-end). The number of characters from src may be different from the number replaced in dst, so the string may grow or contract. The special case where dst-start is equal to dst-end corresponds to insertion; the case where src-start is equal to src-end corresponds to deletion. The order in which characters are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. Returns string, appended with the characters form the concatenation of the given arguments, which can be either strings or characters.

It is guaranteed that string-replace! will return the same object that was passed to it as first argument, whose size may be larger.

ℹ This function is defined in SRFI-118.

R^{7}RS procedure

```
(string-foldcase str)
(string-foldcase str start)
(string-foldcase str start end)
```

Returns a string in which the Unicode simple case-folding algorithm has been applied on `str` between the `start` and `end` indices. If `start` is omited, it defaults to 0. If `end` is omited, it defaults to the length of `str`.

ℹ In R^{7}RS, `string-foldcase` accepts only one argument.

STklos procedure

```
(string-foldcase! str)
(string-foldcase! str start)
(string-foldcase! str start end)
```

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

4.8. 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 2)` in element 1, and the string `"Anna"` in element 2 can be written as following:

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

In STklos, vectors constants don't need to be quoted.

R^{5}RS procedure

```
(vector? obj)
```

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

R^{5}RS procedure

```
(make-vector k)
(make-vector k fill)
```

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.

R^{5}RS procedure

```
(vector obj ...)
```

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

```
(vector 'a 'b 'c)                =>  #(a b c)
```

R^{5}RS procedure

```
(vector-length vector)
```

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

R^{5}RS procedure

```
(vector-ref vector k)
```

`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
```

R^{5}RS procedure

```
(vector-set! vector k obj)
```

`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
```

R^{5}RS / R^{7}RS procedure

```
(vector→list vector)
(vector→list vector start)
(vector→list vector start end)
(list→vector list)
```

Vector→list returns a newly allocated list of the objects contained in the elements of vector between start an end. List→vector returns a newly created vector initialized to the elements of the list list.

In both procedures, order is preserved.

```
(vector->list '#(dah dah didah))     =>  (dah dah didah)
(vector->list '#(dah dah didah) 1 2) =>  (dah)
(list->vector '(dididit dah))        =>  #(dididit dah)
```

The R^{5}RS version of vector→list accepts only one parameter.

R^{7}RS procedure

```
(vector→string string)
(vector→string string start)
(vector→string string start end)
(string→vector vector)
(string→vector vector start)
(string→vector vector start end)
```

The vector→string procedure returns a newly allocated string of the objects contained in the elements of vector between start and end. It is an error if any element of vector between start and end is not a character.

The string→vector procedure returns a newly created vector initialized to the elements of string between start and end.

In both procedures, order is preserved.

```
(string->vector "ABC")           => #(#\A #\B #\C)
(vector->string #(#\1 #\2 #\3))  => "123"
```

R^{5}RS procedure

```
(vector-append vector …)
```

Returns a newly allocated vector whose elements are the concatenation of the elements of the given vectors.

```
(vector-append #(a b c) #(d e f)) => #(a b c d e f)
```

R^5RS / R^7RS procedure

```
(vector-fill! vector fill)
(vector-fill! vector fill start)
(vector-fill! vector fill start end)
```

Stores `fill` in every element of `vector` between `start` and `end`.

The R^5RS version of `vector-fill!` accepts only one parameter.

R^5RS / R^7RS procedure

```
(vector-copy v)
(vector-copy v start)
(vector-copy v start stop)
```

Return a newly allocated copy of the elements of the given vector between `start` and `end` . The elements of the new vector are the same (in the sense of eqv?) as the elements of the old.

Note that, if `v` is a constant vector, its copy is not constant.

```
(define a #(1 8 2 8))           ; a is immutable
(define b (vector-copy a))      ; b is mutable
(vector-set! b 0 3)
b                             => #(3 8 2 8)
(define c (vector-copy b 1 3))
c                             => #(8 2)
```

R^7RS procedure

```
(vector-copy! to at from)
(vector-copy! to at from start)
(vector-copy! to at from start end)
```

STklos procedure

```
(vector-resize v size)
(vector-resize v size fill)
```

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***.

STklos procedure

```
(vector-mutable? obj)
```

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
```

STklos procedure

```
(sort obj predicate)
```

`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")
```

4.9. Structures

A structure type is a record data type composing a number of slots. A structure, an instance of a structure type, is a first-class value that contains a value for each field of the structure type.

Structures can be created with the `define-struct` high level syntax. However, ***STklos*** also offers some low-level functions to build and access the internals of a structure.

STklos syntax

```
(define-struct <name> <slot> …)
```

Defines a structure type whose name is `<name>`. Once a structure type is defined, the following symbols are bound:

- `<name>` denotes the structure type.
- `make-<name>` is a procedure which takes 0 to `n` parameters (if there are `n` slots defined). Each parameter is assigned to the corresponding field (in the definition order).
- `<name>?` is a predicate which returns `#t` when applied to an instance of the `<name>` structure type and `#f` otherwise
- `<name>-<slot>` (one for each defined `<slot>`) to read the content of an instance of the `<name>` structure type. Writting the content of a slot can be done using a generalized `set!`.

```
(define-struct point x y)
(define p (make-point 1 2))
(point? p)     => #t
(point? 100)   => #f
(point-x p)    => 1
(point-y p)    => 2
(set! (point-x p) 10)
(point-x p)    => 10
```

STklos procedure

```
(make-struct-type name parent slots)
```

This form which is more general than `define-struct` permits to define a new structure type whose name is `name`. Parent is the structure type from which is the new structure type is a subtype (or `#f` is the new structure-type has no super type). `Slots` is the list of the slot names which constitute the structure tpe.

When a structure type is s subtype of a previous type, its slots are added to the ones of the super type.

STklos procedure

```
(struct-type? obj)
```

Returns `#t` if `obj` is a structure type, otherwise returns `#f`.

```
(let ((type (make-struct-type 'point #f '(x y))))
  (struct-type? type))         => #t
```

STklos procedure

```
(struct-type-slots structype)
```

Returns the slots of the structure type `structype` as a list.

```
(define point  (make-struct-type 'point #f '(x y)))
(define circle (make-struct-type 'circle point '(r)))
(struct-type-slots point)   => (x y)
(struct-type-slots circle)  => (x y r)
```

STklos procedure

```
(struct-type-parent structype)
```

Returns the super type of the structure type `structype`, if it exists or `#f` otherwise.

STklos procedure

```
(struct-type-name structype)
```

Returns the name associated to the structure type `structype`.

STklos procedure

```
(struct-type-change-writer! structype proc)
```

Change the default writer associated to structures of type `structype` to the `proc` procedure. The `proc` procedure must accept 2 arguments (the structure to write and the port wher the structure must be written in that order). The value returned by `struct-type-change-writer!` is the old writer associated to `structype`. To restore the standard structure writer for `structype`, use the special value `#f`.

```
(define point (make-struct-type 'point #f '(x y)))

(struct-type-change-writer!
  point
  (lambda (s port)
    (let ((type (struct-type s)))
      (format port "{~A" (struct-type-name type))
      ;; display the slots and their value
      (for-each (lambda (x)
       (format port " ~A=~S" x (struct-ref s x)))
     (struct-type-slots type))
      (format port "}"))))
(display (make-struct point 1 2)) |- {point x=1 y=2}
```

STklos procedure

```
(make-struct structype expr …)
```

Returns a newly allocated instance of the structure type `structype`, whose slots are initialized to `expr` ... If fewer `expr` than the number of instances are given to `make-struct`, the remaining slots are inialized with the special ***void*** value.

STklos procedure

```
(struct? obj)
```

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

```
(let* ((type (make-struct-type 'point #f '(x y)))
       (inst (make-struct type 1 2)))
  (struct? inst))         => #t
```

STklos procedure

```
(struct-type s)
```

Returns the structure type of the `s` structure

STklos procedure

```
(struct-ref s slot-name)
```

Returns the value associated to slot `slot-name` of the `s` structure.

```
(define point  (make-struct-type 'point #f '(x y)))
(define circle (make-struct-type 'circle point '(r)))
(define p (make-struct point 1 2))
(define c (make-struct circle 10 20 30))
(struct-ref p 'y) => 2
(struct-ref c 'r) => 30
```

STklos procedure

```
(struct-set! s slot-name value)
```

Stores value in the to slot `slot-name` of the `s` structure. The value returned by `struct-set!` is ***void***.

```
(define point  (make-struct-type 'point #f '(x y)))
(define p (make-struct point 1 2))
(struct-ref p 'x) => 1
(struct-set! p 'x 0)
(struct-ref p 'x) => 0
```

STklos procedure

```
(struct-is-a? s structype)
```

Return a boolean that indicates if the structure `s` is of type `structype`. Note that if `s` is an instance of a subtype of *S*, it is considered also as an instance of type *S*.

```
(define point  (make-struct-type 'point #f '(x y)))
(define circle (make-struct-type 'circle point '(r)))
(define p (make-struct point 1 2))
(define c (make-struct circle 10 20 30))
(struct-is-a? p point)   => #t
(struct-is-a? c point)   => #t
(struct-is-a? p circle)  => #f
(struct-is-a? c circle)  => #t
```

STklos procedure

```
(struct→list s)
```

Returns the content of structure `s` as an A-list whose keys are the slots of the structure type of `s`.

```
(define point  (make-struct-type 'point #f '(x y)))
(define p (make-struct point 1 2))
(struct->list p) => ((x . 1) (y . 2))
```

4.10. Bytevectors

Bytevectors represent blocks of binary data. They are fixed-length sequences of bytes, where a *byte* is an exact integer in the range `(0, 255)`. A bytevector is typically more space-efficient than a vector containing the same values.

The *length* of a bytevector is the number of elements that it contains. This number is a non-negative integer that is fixed when the bytevector is created. The *valid indexes* of a bytevector are the exact non-negative integers less than the length of the bytevector, starting at index zero as with vectors.

Bytevectors are written using the notation `#u8(byte …)`. For example, a bytevector of length 3 containing the byte 0 in element 0, the byte 10 in element 1, and the byte 5 in element 2 can be written as follows: `#u8(0 10 5)`

Bytevector constants are self-evaluating, so they do not need to be quoted in programs.

R^{7}RS procedure

```
(bytevector? obj)
```

Returns `!t` if `obj` is a bytevector and returns `!f` otherwise.

R^{7}RS procedure

```
(make-bytevector k)
(make-bytevector k byte)
```

Returns a newly allocated bytevector of k bytes. If If `byte` is given, then all elements of the bytevector are initialized to `byte`, otherwise the contents of each element is 0.

```
(make-bytevector 2 12) => #u8(12 12)
(make-bytevector 3)     => #u8(0 0 0)
```

R^{7}RS procedure

```
(bytevector byte …)
```

Returns a newly allocated bytevector containing its arguments.

```
(bytevector 1 3 5 1 3 5)    => #u8(1 3 5 1 3 5)
(bytevector)                => #u8()
```

R^{7}RS procedure

```
(bytevector-length bytevector)
```

Returns the length of `bytevector` in bytes as an exact integer.

R^{7}RS procedure

```
(bytevector-copy bytevector)
(bytevector-copy bytevector start)
(bytevector-copy bytevector start end)
```

Returns a newly allocated bytevector containing the bytes in `bytevector` between `start` and `end`.

```
(define a #u8(1 2 3 4 5))
(bytevector-copy a 2 4))       =>  #u8(3 4)
```

R^{7}RS procedure

```
(bytevector-copy! to at from)
(bytevector-copy! to at from start)
(bytevector-copy! to at from start end)
```

Copies the bytes of bytevector `from` between `start` and `end` to bytevector `to`, starting at `at`. The order in which bytes are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary bytevector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if `at` is less than zero or greater than the length of `to`. It is also an error if `(- (bytevector-length to) at)` is less than `(- end start)`.

```
(define a (bytevector 1 2 3 4 5))
(define b (bytevector 10 20 30 40 50))
(bytevector-copy! b 1 a 0 2)
b                                  =>  #u8(10 1 2 40 50
```

R5RS procedure

```
(bytevector-append bytevector …)
```

Returns a newly allocated bytevector whose elements are the concatenation of the elements in the given bytevectors.

```
(bytevector-append #u8(0 1 2) #u8(3 4 5))
                   =>  #u8(0 1 2 3 4 5)
```

R7RS procedure

```
(utf8→string bytevector)
(utf8→string bytevector start)
(utf8→string bytevector start end)
(string→utf8 string)
(string→utf8 string start)
(string→utf8 string start end)
```

These procedures translate between strings and bytevectors that encode those strings using the UTF-8 encoding. The `utf8→string` procedure decodes the bytes of a bytevector between `start` and `end` and returns the corresponding string; the `string→utf8` procedure encodes the characters of a string between `start` and `end` and returns the corresponding bytevector.

It is an error for `bytevector` to contain invalid UTF-8 byte sequences.

```
(utf8->string #u8(#x41))    => "A"
(string->utf8 "λ")          => #u8((#xce #xbb)
```

4.11. Control features

R^{5}RS procedure

```
(procedure? obj)
```

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
```

R^{5}RS procedure

```
(apply proc arg1 … args)
```

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
```

R^{5}RS procedure

```
(map proc list1 list2 …)
```

The list`s 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 `list`s 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) ,(emph "or") (2 1)
```

R^{7}RS procedure

```
(string-map proc string1 string2 …)
```

The strings must be strings, and proc must be a procedure taking as many arguments as there are strings and returning a single value. If more than one string is given and not all strings have the same length, string-map terminates when the shortest list runs out. String-map applies proc element-wise to the elements of the strings and returns a string of the results, in order. The dynamic order in which proc is applied to the elements of the strings is unspecified.

```
(string-map char-downcase "AbdEgH")
         => "abdegh"

(string-map
  (lambda (c)
    (integer->char (+ 1 (char->integer c))))
  "HAL")
         => "IBM"

(string-map (lambda (c k)
           (if (eqv? k #\u)
               (char-upcase c)
               (char-downcase c)))
         "studlycaps"
         "ululululul")
      => "StUdLyCaPs"
```

R^{7}RS procedure

```
(vector-map proc vector1 vector2 ⋯)
```

The vectors must be vectors, and proc must be a procedure taking as many arguments as there are vectors and returning a single value. If more than one vector is given and not all vectors have the same length, vector-map terminates when the shortest list runs out. Vector-map applies proc element-wise to the elements of the vectors and returns a vector of the results, in order. The dynamic order in which proc is applied to the elements of the vectors is unspecified.

```
(vector-map cadr '#((a b) (d e) (g h)))
    =>  #(b e h)

(vector-map (lambda (n) (expt n n))
         '#(1 2 3 4 5))
    => #(1 4 27 256 3125)

(vector-map + '#(1 2 3) '#(4 5 6))
    => #(5 7 9)

(let ((count 0))
  (vector-map
    (lambda (ignored)
      (set! count (+ count 1))
      count)
    '#(a b)))
    => #(1 2) or #(2 1)
```

R^{5}RS procedure

```
(for-each proc list1 list2 ⋯)
```

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)
```

R^{7}RS procedure

```
(string-for-each proc string1 string2 ...)
```

The arguments to `string-for-each` are like the arguments to `string-map`, but `string-for-each` calls `proc` for its side effects rather than for its values. Unlike `string-map`, `string-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 `string-for-each` is unspecified. If more than one string is given and not all strings have the same length, `string-for-each` terminates when the shortest string runs out.

```
(let ((v (list)))
  (string-for-each (lambda (c) (set! v (cons (char->integer c) v)))
                   "abcde")
   v)
       => (101 100 99 98 97)
```

R^7RS procedure

```
(vector-for-each proc vector1 vector2 ...)
```

The arguments to `vector-for-each` are like the arguments to `vector-map`, but `vector-for-each` calls `proc` for its side effects rather than for its values. Unlike `vector-map`, `vector-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 `vector-for-each` is unspecified. If more than one vector is given and not all vectors have the same length, `vector-for-each` terminates when the shortest vector runs out.

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

STklos procedure

```
(every pred list1 list2 ...)
```

`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.

STklos procedure

```
(any pred list1 list2 ...)
```

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
```

R^{5}RS procedure

```
(call-with-current-continuation proc)
(call/cc proc)
```

Proc must be a procedure of one argument. The procedure call-with-current-continuation packages up the current continuation (see the rationale below) as an "*escape procedure*" and passes it as an argument to proc. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. Calling the escape procedure may cause the invocation of before and after thunks installed using dynamic-wind.

The escape procedure accepts the same number of arguments as the continuation to the original call to call-with-current-continuation. Except for continuations created by the call-with-values procedure, all continuations take exactly one value.

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

The following examples show only the most common ways in which call-with-current-continuation is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                                 =>  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))                 =>  4
(list-length '(a b . c))                 =>  #f
```

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

Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. Call-with-current-continuation allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

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

STklos procedure

```
(call/ec proc)
```

`call/ec` is an short name for `call-with-escape-continuation`. `call/ec` calls `proc` with one parameter, which is the current escape continuation (a continuation which can only be used to abort a computation and hence cannot be "re-enterered".

```
(list 1
      (call/ec (lambda (return) (list 'a (return 'b) 'c)))
      3)        => (1 b 3)
```

`call/ec` is cheaper than the full call/cc. It is particularily useful when all the power of `call/cc` is not neededd.

R^{5}RS procedure

```
(values obj …)
```

Delivers all of its arguments to its continuation.

R^{5}RS imposes to use multiple values in the context 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*)).

R^{5}RS procedure

```
(call-with-values producer consumer)
```

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
```

STklos syntax

```
(receive <formals> <expression> <body>)
```

This form is defined in **SRFI-8** (*Receive: 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)
```

R^{5}RS procedure

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

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. It is defined as follows:

- The dynamic extent is entered when execution of the body of the called procedure begins.
- The dynamic extent is also entered when execution is not within the dynamic extent and a continuation is invoked that was captured (using `call-with-current-continuation`) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to `thunk` and then a continuation is invoked in such a way that the afters from these two invocations of `dynamic-wind` are both to be called, then the after associated with the second (inner) call to `dynamic-wind` is called first.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to `thunk` and then a continuation is invoked in such a way that the befores from these two invocations of `dynamic-wind` are both to be called, then the before associated with the first (outer) call to `dynamic-wind` is called first.

If invoking a continuation requires calling the `before` from one call to `dynamic-wind` and the `after` from another, then the `after` is called first.

The effect of using a captured continuation to enter or exit the dynamic extent of a call to `before` or `after` is undefined.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))
                 =>  (connect talk1 disconnect
                      connect talk2 disconnect)
```

R^{5}RS procedure

```
(eval expression environment)
(eval expression)
```

Evaluates expression in the specified environment and returns its value. `Expression` must be a valid Scheme expression represented as data. `Environment` may be a R^{5}RS environment-specifier (`interaction-environment`, `scheme-report-environment` or `null-environment`) or a ***STklos*** module.

```
(eval '(* 7 3) (scheme-report-environment 5))
              => 21
(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
              => 20
(define-module A
  (define x 1))
(eval '(cons x x) (find-module 'A))
              => (1 . 1)
```

R^{7}RS procedure

```
(environment set1 ...)
```

This procedure returns a specifier for the environment that results by starting with an empty environment and then importing each **set**, considered as an import set, into it. The bindings of the environment represented by the specifier, as is the environment itself.

In STklos, - each `set` argument can be a list (specifying an R^{7}RS library) or a symbol (specifying a module). - the return environment is an R^{7}RS library (which can be passed to `eval`).

```
(eval '(* 7 3) (environment '(scheme base)))    => 21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))                                     => 20

(eval '(define foo 32)
      (environment '(scheme base)))             => error

(let ((e (environment '(only (scheme base) + -)
                      '(only (scheme write) display))))
  (length (module-symbols e)))                  => 3

(let ((e (environment '(prefix (only (scheme base) car)
                               foo-))))
```

```
    (module-symbols e))                            => (foo-car)
```

R^{5}RS procedure

```
(scheme-report-environment)
(scheme-report-environment version)
```

Returns a specifier for an environment that contains the bindings defined in the R^{5}RS report.

> In STklos, `scheme-report-environment` function can be called without the version number (defaults to 5).

R^{5}RS procedure

```
(null-environment)
(null-environment version)
```

Returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in the R^{5}RS report.

> In STklos, `null-environment` function can be called without the version number (defaults to 5).

R^{5}RS procedure

```
(interaction-environment)
```

This procedure returns the environment in the expression are evaluated by default (the ***STklos*** module). The returned environment is mutable.

STklos procedure

```
(eval-from-string str)
(eval-from-string str module)
```

Read an expression from `str` and evaluates it with `eval`. If a `module` is passed, the evaluation takes place in the environment of this module. Otherwise, the evaluation takes place in the environment returned by `current-module`.

```
(define x 10)
(define-module M
  (define x 100))
(eval-from-string "(+ x x)")                   => 20
(eval-from-string "(+ x x)" (find-module 'M))  => 200
```

4.12. Input and Output

R^{5}RS 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 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 `"| "` (that is a pipe bar followed by a space). Specification of a pipe port can occur everywhere a file name is needed.
- Virtual ports are created by supplying basic I/O functions at port creation time. These functions will be used to simulate low level accesses to a ``virtual device''. This kind of port is particularly convenient for reading or writing in a graphical window as if it was a file. Once a virtual port is created, it can be accessed as a normal port with the standard Scheme primitives.

4.12.1. Ports

R^{7}RS procedure

```
(call-with-port port proc)
```

The `call-with-port` procedure calls `proc` with `port` as an argument. If `proc` returns, then the `port` is closed automatically and the values yielded by the `proc` are returned. If `proc` does not return, then the `port` must not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

It is an error if proc does not accept one argument.

R^{5}RS procedure

```
(call-with-input-file string proc)
(call-with-output-file string proc)
```

`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.

> 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`.

STklos procedure

```
(call-with-input-string string proc)
```

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

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

STklos procedure

```
(call-with-output-string proc)
```

`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"
```

R^{5}RS procedure

```
(input-port? obj)
```

```
(output-port? obj)
```

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

R^{7}RS procedure

```
(textual-port? obj)
(binary-port? obj)
```

Returns #t if obj is a textual port or binary port respectively, otherwise returns #f.

R^{7}RS procedure

```
(port? obj)
```

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

STklos procedure

```
(input-string-port? obj)
(output-string-port? obj)
```

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

STklos procedure

```
(input-bytevector-port? obj)
(output-bytevector-port? obj)
```

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

STklos procedure

```
(input-file-port? obj)
(output-file-port? obj)
```

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

STklos procedure

```
(input-virtual-port? obj)
(output-virtual-port? obj)
```

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

STklos procedure

```
(interactive-port? port)
```

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

R^{5}RS procedure

```
(current-input-port obj)
(current-output-port obj)
```

Returns the current default input or output port.

STklos procedure

```
(current-error-port obj)
```

Returns the current default error port.

R^{5}RS procedure

```
(with-input-from-file string thunk)
(with-output-to-file string thunk)
```

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 ,(emph "ls") command (i.e. the output of the ,(emph "ls") command is ,(emph "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 Scheme")
    (newline)))
```

STklos procedure

```
(with-error-to-file string thunk)
```

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

STklos procedure

```
(with-input-from-string string thunk)
```

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
```

STklos procedure

```
(with-output-to-string thunk)
```

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\\""
```

STklos procedure

```
(with-input-from-port port thunk)
(with-output-to-port port thunk)
(with-error-to-port port thunk)
```

`Port` should be a port, and `proc` should be a procedure of no arguments. These procedures do a job similar to the `with-⋯-file` counterparts excepted that they use an open port instead of string specifying a file name

R^{5}RS procedure

```
(open-input-file filename)
```

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.

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.

STklos procedure

```
(open-input-string str)
```

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

R7RS procedure

```
(open-input-string bytevector)
```

Takes a bytevector and returns a binary input port that delivers bytes from the `bytevector`.

STklos procedure

```
(open-input-virtual :key (read-char #f) (ready? #f) (eof? #f) (close #f))
```

Returns a virtual port using the `read-char` procedure to read a character from the port, `ready?` to know if there is any data to read from the port, `eof?` to know if the end of file is reached on the port and finally `close` to close the port. All these procedure takes one parameter which is the port from which the input takes place. `Open-input-virtual` accepts also the special value `#f` for the I/O procedures with the following conventions:

- if `read-char` or `eof?` is `#f`, any attempt to read the virtual port will return an eof object;
- if `ready?` is `#f`, the file is always ready for reading;
- if `close` is `#f`, no action is done when the port is closed.

Hereafter is a possible implementation of `open-input-string` using virtual ports:

```
(define (open-input-string str)
  (let ((index 0))
    (open-input-virtual
       :read-char (lambda (p)
                    ;; test on eof is already done by the system
                    (let ((res (string-ref str index)))
                      (set! index (+ index 1))
                      res))
       :eof? (lambda (p) (>= index (string-length str))))))
```

R5RS procedure

```
(open-output-file filename)
```

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.

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.

STklos procedure

```
(open-output-string)
```

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

R7RS procedure

```
(open-output-bytevector)
```

Returns a binary output port that will accumulate bytes for retrieval by `get-output-bytevector`.

STklos procedure

```
(open-output-virtual :key (write-char #f) (write-string #f) (flush #f) (close #f))
```

Returns a virtual port using the `write-char` procedure to write a character to the port, `write-string` to write a string to the port, `flush` to (eventuelly) flush the characters on the port and finally `` close`to close the port. `Write-char `` takes two parameters: a character and the port to which the output must be done. `write-string` takes two parameters: a string and a port. `Flush` and `Close` take one parameter which is the port on which the action must be done. `Open-output-virtual` accepts also the special value `#f` for the I/O procedures. If a procedure is `#f` nothing is done on the corresponding action.

Hereafter is a (very inefficient) implementation of a variant of `open-output-string` using virtual ports. The value of the output string is printed when the port is closed:

```
(define (open-output-string)
  (let ((str ""))
    (open-output-virtual
       :write-char (lambda (c p)
                     (set! str (string-append str (string c))))
       :write-string (lambda (s p)
                       (set! str (string-append str s)))
       :close (lambda (p) (write str) (newline)))))
```

`write-string` is mainly used for writing strings and is generally more efficient than writing the string character by character. However, if `write-string` is not provided,

> strings are printed with `write-char`. On the other hand, if `write-char` is absent, characters are written by successive allocation of one character strings.

Hereafter is another example: a virtual file port where all characters are converted to upper case:

```
(define (open-output-uppercase-file file)
  (let ((out (open-file file "w")))
    (and out
         (open-output-virtual
             :write-string (lambda (s p)
                             (display (string-upper s) out))
             :close (lambda (p)
                      (close-port out))))))
```

STklos procedure

```
(open-file filename mode)
```

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 textual 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"`).

STklos procedure

```
(get-output-string port)
```

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"
```

STklos procedure

```
(get-output-bytevector port)
```

Returns a bytevector consisting of the bytes that have been output to the `port` so far in the order they were output.

```
(let ((p (open-output-bytevector)))
   (u8-write 65)
   (u8-write 66)
   (get-output-bytevector p))          => #u8(65 66)
```

R^{5}RS procedure

```
(close-input-port port)
(close-output-port port)
```

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***.

R^{7}RS procedure

```
(close-port port)
```

Closes the port associated with `port`.

STklos procedure

```
(port-rewind port)
```

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

STklos procedure

```
(port-seek port pos)
(port-seek port pos whence)
```

Sets the file position for the given `port` to the position `pos`. The new position, measured in bytes, is obtained by adding `pos` bytes to the position specified by `whence`. If passed, `whence` must be one of `:start`, `:current` or `:end`. The resulting position is relative to the start of the file, the current position indicator, or end-of-file, respectively. If `whence` is omitted, it defaults to `:start`.

> After using port-seek, the value returned by `port-current-line` may be incorrect.

STklos procedure

```
(port-current-line)
(port-current-line port)
```

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`.

> The `port-seek`, `read-chars` and `read-chars!` procedures generally break the line-number. After using one of these procedures, the value returned by `port-current-line` will be `-1` (except a `port-seek` at the beginning of the port reinitializes the line counter).

STklos procedure

```
(port-current-position)
(port-current-position port)
```

Returns the position associated to the given `port` as an integer (i.e. number of characters from the beginning of the port). The `port` argument may be omitted, in which case it defaults to the value returned by `current-input-port`.

STklos procedure

```
(port-file-name port)
```

Returns the file name used to open `port`; `port` must be a file port.

STklos procedure

```
(port-idle-register! port thunk)
(port-idle-unregister! port thunk)
(port-idle-reset! port)
```

`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 ***sexpr*** 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))
```

STklos procedure

```
(port-closed? port)
(port-open? port)
```

`port-closed?` returns `#t` if `port` is closed and `#f` otherwise. On the contrary, `port-open?` returns `#t` if `port` is open and `#f` otherwise.

`port-closed?` was the usual STklos function to test if a port is closed. `port-open?` has been added to be the companion of the R^{7}RS functions `input-port-open?` and `output-port-open?`

STklos procedure

```
(port-close-hook-set! port thunk)
```

Associate the procedure `thunk` to `port`. The thunk will be called the first time `port` is closed.

```
(let* ((tmp (temporary-file-name))
       (p   (open-output-file tmp))
```

```
      (foo #t))
  (port-close-hook-set! p
                       (lambda()
                         (remove-file tmp)
                         (set! foo #t)))
  (close-port p)
  foo)
```

STklos procedure

```
(port-close-hook port)
```

Returns the user close procedure associated to the given `port`.

```
 The following procedures are defined in *link:http://srfi.schemers.org/srfi-192/srfi-
192.html[SRFI-192]* (_Port Positioning_)(((SRFI-192))) which is fully
  supported:((("SRFI-192")))
(((port-has-port-position?)))
```

STklos procedure

```
(port-has-port-position? port)
```

The port-has-port-position? procedure returns #t if the port supports the port-position operation, and #f otherwise. If the port does not support the operation, port-position signals an error.

STklos procedure

```
(port-position port)
```

The port-position procedure returns an object representing the information state about the port current position as is necessary to save and restore that position. This value can be useful only as the pos argument to set-port-position!, if the latter is even supported on the port. However, if the port is binary and the object is an exact integer, then it is the position measured in bytes, and can be used to compute a new position some specified number of bytes away.

STklos procedure

```
(port-has-set-port-position!? port)
```

The port-has-set-port-position!? procedure returns #t if the port supports the set-port-position! operation, and #f otherwise.

STklos procedure

```
(set-port-position! port pos)
```

For a textual port, it is implementation-defined what happens if pos is not the return value of a call to port-position on port. However, a binary port will also accept an exact integer, in which case the port position is set to the specified number of bytes from the beginning of the port data. If this is not sufficient information to specify the port state, or the specified position is uninterpretable by the port, an error satisfying i/o-invalid-position-error? is signaled.

If `set-port-position!` procedure is invoked on a port that does not support the operation or if pos is not in the range of valid positions of port, `set-port-position!` signals an error. Otherwise, it sets the current position of the port to pos. If port is an output port, `set-port-position!` first flushes port (even if the port position will not change).

If port is a binary output port and the current position is set beyond the current end of the data in the underlying data sink, the object is not extended until new data is written at that position. The contents of any intervening positions are unspecified. It is also possible to set the position of a binary input port beyond the end of the data in the data source, but a read will fail unless the data has been extended by other means. File ports can always be extended in this manner within the limits of the underlying operating system. In other types of ports, if an attempt is made to set the position beyond the current end of data in the underlying object, and the object does not support extension, an error satisfying `i/o-invalid-position-error?` is signaled.

STklos procedure

```
(make-i/o-invalid-position-error pos)
```

Returns a condition object which satisfies i/o-invalid-position-error?. The pos argument represents a position passed to set-position!.

STklos procedure

```
(i/o-invalid-position-error? obj)
```

Returns `#t` if `obj` is an object created by `make-i/o-invalid-position-error?` or an object raised in the circumstances described in **SRFI-192** (attempt to access an invalid position in the stream), or `#f` if it is not.

4.12.2. Input

R^{5}RS procedure

```
(read)
(read port)
```

`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.

STklos `read` supports the **SRFI-10** `#,()` form that can be used to denote values that do not have a convenient printed representation. See the SRFI document for more information.

STklos procedure

```
(read-with-shared-structure)
(read-with-shared-structure port)
(read/ss)
(read/ss port)
```

`read-with-shared-structure` is identical to `read`. It has been added to be compatible with ,(link-srfi 38). STklos always knew how to deal with recursive input data. `read/ss` is only a shorter name for `read-with-shared-structure`.

STklos procedure

```
(define-reader-ctor tag proc)
```

This procedure permits to define a new user to reader constructor procedure at run-time. It is defined in ,(link-srfi 10) document. See SRFI document for more information.

```
(define-reader-ctor 'rev (lambda (x y) (cons y x)))
```

```
(with-input-from-string "#,(rev 1 2)" read)
                              => (2 . 1)
```

R^{5}RS procedure

```
(read-char)
(read-char port)
```

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`.

STklos procedure

```
(read-bytes size)
(read-bytes size port)
```

Returns a newly allocated string made of `size` characters read from `port`. If less than `size` characters are available on the input port, the returned string is smaller than `size` and its size is the number of available characters. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

This function was previously called `read-chars`. Usage of the old name is deprecated.

R^{7}RS procedure

```
(read-bytevector k)
(read-bytevector k port)
```

Reads the next `k` bytes, or as many as are available before the end of file, from the textual input `port` into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

R^{7}RS procedure

```
(read-bytevector! k)
(read-bytevector! k port)
(read-bytevector! k port start)
```

```
(read-bytevector! k port start end)
```

Reads the next `end - start` bytes, or as many as are available before the end of file, from the binary input port into `bytevector` in left-to-right order beginning at the start position. If `end` is not supplied, reads until the end of `bytevector` has been reached. If `start` is not supplied, reads beginning at position 0. Returns the number of bytes read. If no bytes are available, an end-of-file object is returned.

STklos procedure

```
(read-bytes! str)
(read-bytes! str port)
```

This function reads the characters available from `port` in the string `str` by chuncks whose size is equal to the length of `str`. The value returned by `read-bytes!` is an integer indicating the number of characters read. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

This function is similar to `read-bytes` except that it avoids to allocate a new string for each read.

```
(define (copy-file from to)
  (let* ((size 1024)
         (in  (open-input-file from))
         (out (open-output-file to))
         (s   (make-string size)))
    (let Loop ()
      (let ((n (read-bytes! s in)))
        (cond
          ((= n size)
             (write-chars s out)
             (Loop))
          (else
             (write-chars (substring s 0 n) out)
             (close-port out)))))))
```

This function was previously called `read-chars!`. Usage of the old name is deprecated.

STklos procedure

```
(read-byte)
(read-byte port)
```

Returns the next character available from the input port as an integer. If the end of file is reached, this function returns the end of file object.

R^{5}RS procedure

```
(peek-char)
(peek-char port)
```

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.

> 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.

STklos procedure

```
(peek-byte)
(peek-byte port)
```

Returns the next character available from the input port, without updating the port to point to the following character. Whereas peek-char returns a character, this function returns an integer between 0and 255.

R^{5}RS procedure

```
(eof-object? obj)
```

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

STklos procedure

```
(eof-object)
```

end of file Returns an end of file object. Note that the special notation #eof is another way to return such an end of file object.

R^{5}RS procedure

```
(char-ready?)
(char-ready? port)
```

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`.

R^{7}RS procedure

```
(read-string k)
(read-string k port)
```

Reads the next `k` characters, or as many as are available before the end of file, from the textual input `port` into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

R^{7}RS procedure

```
(read-u8)
(read-u8 port)
```

Returns the next byte available from the binary input `port`, updating the `port` to point to the following byte. If no more bytes are available, an end-of-file object is returned.

This function is similar to the `read-byte` function, excepted that it can be used only on a binary port.

R^{7}RS procedure

```
(peek-u8)
(peek-u8 port)
```

Returns the next byte available from the binary input `port`, but without updating the `port` to point to the following byte. If no more bytes are available, an end-of-file object is returned.

This function is similar to the `peek-byte` function, excepted that it can be used only on a binary port.

R7RS procedure

```
(u8-ready?)
(u8-ready? port)
```

Returns `#t` if a byte is ready on the binary input `port` and returns `#f` otherwise. If `u8-ready?` returns `#t` then the next read-u8 operation on the given port is guaranteed not to hang. If the `port` is at end of file then `u8-ready?` returns `#t`.

STklos procedure

```
(read-line)
(read-line port)
```

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`.

As said in *primitive* `values`, if `read-line` is not used in the context of `call-with-values`, the second value returned by this procedure is ignored.

STklos procedure

```
(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
```

STklos procedure

```
(port->string port)
```

```
(port→sexp-list port)
(port→string-list port)
```

All these procedures 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\"")
```

4.12.3. Output

R⁵RS procedure

```
(write obj)
(write obj port)
```

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 ,(emph "#\\") 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`.

R⁷RS procedure

```
(write-shared obj)
(write-shared obj port)
```

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 Section 1.2.4).

This function is also called `write*`. The name `write*` was the name used by ***STklos*** for `write-shared` before it was introduced in R⁷RS.

STklos procedure

```
(write-with-shared-structure obj)
(write-with-shared-structure obj port)
```

```
(write-with-shared-structure obj port optarg)
(write/ss obj)
(write/ss obj port)
(write/ss obj port optarg)
```

`write-with-shared-structure` has been added to be compatible with **SRFI-38** (*External representation of shared structures*). It is is identical to `write*`, except that it accepts one more parameter (`optarg`). This parameter, which is not specified in **SRFI-38**, is always ignored. `write/ss` is only a shorter name for `write-with-shared-structure`.

R^{5}RS procedure

```
(display obj)
(display obj port)
```

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`.

`Write` is intended for producing machine-readable output and `display` is for producing human-readable output.

As required by R^{7}RS does not loop forever when `obj` contains self-references.

STklos procedure

```
(display-shared obj)
(display-shared obj port)
```

The `display-shared` procedure is the same as `display`, except that shared structure are represented using datum labels.

STklos procedure

```
(display-simple obj)
(display-simple obj port)
```

The `display-simple` procedure is the same as `display`, except that shared structure is never represented using datum labels. This can cause `display-simple` not to terminate if `obj` contains circular structure.

R^{5}RS procedure

```
(newline)
(newline port)
```

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`.

R^{7}RS procedure

```
(write-string string)
(write-string string port)
(write-string string port start)
(write-string string port start end)
```

Writes the characters of `string` from `start` to `end` in left-to-right order to the textual output `port`.

R^{7}RS procedure

```
(write-u8 byte)
(write-u8 byte port)
```

Writes the `byte` to the given binary output port.

R^{7}RS procedure

```
(write-bytevector bytevector)
(write-bytevector bytevector port)
(write-bytevector bytevector port start)
(write-bytevector bytevector port start end)
```

Writes the bytes of `bytevector` from `start` to `end` in left-to-right order to the binary output `port`.

R^{5}RS procedure

```
(write-char char)
```

```
(write-char char port)
```

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`.

STklos procedure

```
(write-chars str)
(write-chars str port)
```

Writes the characters of string `str` 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`.

> This function is generally faster than `display` for strings. Furthermore, this primitive does not use the buffer associated to `port`.

STklos procedure

```
(write-byte b)
(write-byte b port)
```

Write byte `b` to the port. `b` must be an exact integer in range between 0 and 255.

STklos procedure

```
(format port str obj ...)
(format str obj)
```

Writes the `` obj`s 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 `write*`).
- `~d` or `~D` is replaced by the decimal printed representation of the next `obj` (which must be a number).

- ~x or ~X is replaced by the hexadecimal printed representation of the next obj (which must be a number).
- ~o or ~O is replaced by the octal printed representation of the next obj (which must be a number).
- ~b or ~B is replaced by the binary printed representation of the next obj (which must be a number).
- ~c or ~C is replaced by the printed representation of the next obj (which must be a character).
- ~y or ~Y is replaced by the pretty-printed representation of the next obj. The standard pretty-printer is used here.
- ~? is replaced by the result of the recursive call of format with the two next obj: the first item should be a string, and the second, a list with the arguments.
- ~k or ~K is another name for ~?
- ~[w[,d]]f or ~[w[,d]]F is replaced by the printed representation of next obj (which must be a number) with width w and d digits after the decimal. Eventually, d may be omitted.
- ~~ is replaced by a single tilde character.
- ~% is replaced by a newline
- ~t or ~T is replaced by a tabulation character.
- ~& is replaced by a newline character if it is known that the previous character was not a newline
- ~_ is replaced by a space
- ~h or ~H provides some help

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\"."
(format "~8,2F" 1/3)         => "    0.33"
(format "~6F" 32)            => "    32"
(format "~1,2F" 4321)        => "4321.00"
(format "~1,2F" (sqrt -3.9)) => "0.00+1.97i"
(format "#d~d #x~x #o~o #b~b~%" 32 32 32 32)
                             => "#d32 #x20 #o40 #b100000\n"
(format #f "~&1~&~&2~&~&~&3~%")
                             => "\n1\n2\n3\n"
(format "~a ~? ~a" 'a "~s" '(new) 'test)
                             => "a new test"
```

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

Since version 0.58, `format` is also compliant with **SRFI-48** (*Intermediate Format Strings*).

STklos procedure

```
(flush-output-port)
(flush-output-port port)
```

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`

STklos procedure

```
(print obj …)
(printerr obj …)
```

These procedures display all their arguments followed by a newline. The procedure `print` uses the standard output port, whereas `printerr` uses the current error port

STklos procedure

```
(printf fmt obj …)
(fprintf port fmt obj …)
(eprintf fmt obj …)
```

These procedures are specialized versions of `format` *primitive*. In these procedures, `fmt` is a string using the `format` conventions. `printf` outputs go on the current output port. `fprintf` outputs go on the specified `port`. `eprintf` outputs go on the current error port (note that eprintf always flushes the characters printed).

4.13. System interface

The ***STklos*** system interface offers all the functions defined in R^{7}RS. Note, that the base implementation provides also a subset of the functions defined in **SRFI-170** (*POSIX API*). These functions are described here.

Note, however that **SRFI-170** is fully supported and accessing the other functions it defines can be done by requiring it, as the other SRFIs that STklos supports.

4.13.1. Loading code

R^{5}RS procedure

```
(load filename)
```

`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 `"load-path"` with the suffixes given by `"load-suffixes"`.

STklos procedure

```
(try-load filename)
```

`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` retuns `#f`.

STklos procedure

```
(find-path str)
(find-path str path)
(find-path str path suffixes)
```

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"
```

STklos procedure

```
(current-loading-file)
```

Returns the path of the file that is currently being load.

STklos procedure

```
(require string)
(provide string)
(require/provide string)
(provided? string)
```

`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. `Require/provide` is more or less equivalent to a `require` followed by a `provide`. `Provided?` returns `#t` if `string` was already provided; it returns `#f` otherwise.

4.13.2. File Primitives

STklos procedure

```
(temp-file-prefix)
(temp-file-prefix value)
```

This parameter object permits to change the default prefix used to build temporary file name. Its default value is built using the `TMPDIR` environment variable (if it is defined) and the current process ID. If a value is provided, it must be a string designating a valid prefix path.

This parameter object is also defined in **SRFI-170** (*POSIX API*).

STklos procedure

```
(create-temp-file)
(create-temp-file prefix)
```

Creates a new temporary file and returns two values: its name and an opened file port on it. The optional argument specifies the filename prefix to use, and defaults to the result of invoking `temp-file-prefix`. The returned file port is opened in read/write mode. It ensures that the name cannot be reused by another process before being used in the program that calls `create-temp-file`. Note, that if the opened port is not used, it can be closed and collected by the GC.

```
(let-values (((name port) (create-temp-file)))
  (let ((msg (format "Name: ~s\n" name)))
    (display msg)
    (display msg port)
    (close-port port)))     => prints the name of the temp. file on the
                               current output port and in the file itself.
```

This function is also defined in **SRFI-170** (*POSIX API*).However, in SRFI-170, `create-temp-file` returns only the name of the temporary file.

`temporary-file-name` is another name for this function.

STklos procedure

```
(create-temp-directory)
(create-temp-directory prefix)
```

Creates a new temporary directory and returns its name as a string. The optional argument specifies the filename prefix to use, and defaults to the result of invoking `temp-file-prefix`.

STklos procedure

```
(rename-file string1 string2)
```

Renames the file whose path-name is `string1` to a file whose path-name is `string2`. The result of `rename-file` is ***void***.

This function is also defined in **SRFI-170** (*POSIX API*).

R^{7}RS procedure

```
(delete-file string)
```

Removes the file whose path name is given in `string`. The result of `delete-file` is ***void***.

This function is also called `remove-file` for compatibility reasons. ,(index "remove-file")

STklos procedure

```
(copy-file string1 string2)
```

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***.

STklos procedure

```
(copy-port in out)
(copy-port in out max)
```

Copy the content of port `in`, which must be opened for reading, on port `out`, which must be opened for writing. If `max` is not specified, All the characters from the input port are copied on ouput port. If `max` is specified, it must be an integer indicating the maximum number of characters which are copied from `in` to `out`.

R^{7}RS procedure

```
(file-exists? string)
```

Returns `#t` if the path name given in `string` denotes an existing file; returns `#f` otherwise.

STklos procedure

```
(file-is-directory? string)
(file-is-regular? string)
(file-is-readable? string)
(file-is-writable? string)
(file-is-executable? string)
```

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).

STklos procedure

```
(file-size string)
```

Returns the size of the file whose path name is given in `string`. If `string` denotes a file which does not exist, `file-size` returns `#f`.

STklos procedure

```
(getcwd)
```

Returns a string containing the current working directory.

STklos procedure

```
(chmod str)
(chmod str option1 ...)
```

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)
```

STklos procedure

```
(chdir dir)
```

Changes the current directory to the directory given in string `dir`.

STklos procedure

```
(create-directory dir)
(create-directory dir permissions)
```

Create a directory with name `dir`. If `permissions` is omitted, it defaults to #o775 (masked by the current umask).

This function is also defined in **SRFI-170** (*POSIX API*). The old name `make-directory` is deprecated.

STklos procedure

```
(create-directories dir)
(create-directories dir permissions)
```

Create a directory with name `dir`. No error is signaled if `dir` already exists. Parent directories of `dir` are created as needed. If `permissions` is omitted, it defaults to #o775 (masked by the current umask).

This function was also called `make-directories`. This old name is obsolete.

STklos procedure

```
(ensure-directories-exist path)
```

Create a directory with name `dir` (and its parent directories if needed), if it does not exist yet.

STklos procedure

```
(delete-directory dir)
(remove-directory dir)
```

Delete the directory with name `dir`.

This function is also defined in **SRFI-170** (*POSIX API*). The name `remove-directory` is kept for compatibility.

STklos procedure

```
(directory-files path)
(directory-files path dotfiles?)
```

Returns the list of the files in the directory `path`. The `dotfiles?` flag (default `#f`) causes files beginning with ,(q ".") to be included in the list. Regardless of the value of `dotfiles?`, the two files ,(q ".") and ,(q "..") are never returned.

This function is also defined in **SRFI-170** (*POSIX API*).

STklos procedure

```
(expand-file-name path)
```

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"
```

STklos procedure

```
(canonical-file-name path)
```

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.

STklos procedure

```
(decompose-file-name string)
```

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")
```

STklos procedure

```
(winify-file-name fn)
```

On Win32 system, when compiled with the Cygwin environment, file names are internally represented in a POSIX-like internal form. Winify-file-bame permits to obtain back the Win32 name of an interned file name

```
(winify-file-name "/tmp")
    => "C:\\cygwin\\tmp"
(list (getcwd) (winify-file-name (getcwd)))
    => ("//saxo/homes/eg/Projects/STklos"
        "\\\\saxo\\homes\\eg\\Projects\\STklos")
```

STklos procedure

```
(posixify-file-name fn)
```

On Win32 system, when compiled with the Cygwin environment, file names are internally represented in a POSIX-like internal form. `posixify-file-bame` permits to obtain the interned file name from its external form. file name

```
(posixify-file-name "C:\\cygwin\\tmp")
       => "/tmp"
```

STklos procedure

```
(basename str)
```

Returns a string containing the last component of the path name given in `str`.

```
(basename "/a/b/c.stk") => "c.stk"
```

STklos procedure

```
(dirname str)
```

Returns a string containing all but the last component of the path name given in `str`.

```
(dirname "/a/b/c.stk") => "/a/b"
```

STklos procedure

```
(file-suffix pathname)
```

Returns the suffix of given `pathname`. If no suffix is found, `file-suffix` returns #f.

```
(file-suffix "./foo.tar.gz") => "gz"
(file-suffix "./a.b/c")      => #f
(file-suffix "./a.b/c.")     => ""
(file-suffix "~/.profile")   => #f
```

STklos procedure

```
(file-prefix pathname)
```

Returns the prefix of given `pathname`.

```
(file-prefix "./foo.tar.gz") => "./foo.tar"
(file-prefix "./a.b/c")      => "./a.b/c"
```

STklos procedure

```
(file-separator)
```

Retuns the operating system file separator as a character. This is typically `#\/` on Unix (or Cygwin) systems and `#\\` on Windows.

STklos procedure

```
(make-path dirname . names)
```

Builds a file name from the directory `dirname` and `names`. For instance, on a Unix system:

```
(make-path "a" "b" "c")   => "a/b/c"
```

STklos procedure

```
(glob pattern …)
```

`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.
- `\x` Matches the character `x`.
- `{a,b,…}` Matches any of the strings `a`, `b`, etc.)

As with csh, 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 csh globbing in two ways:

1. it does not sort its result list (use the `sort` procedure if you want the list sorted).
2. `glob` only returns the names of files that actually exist; in csh no check for existence is made unless a pattern contains a `?`, `*`, or `[]` construct.

STklos procedure

```
(posix-error? obj)
```

This procedure returns `#t` if `obj` is a condition object that describes a POSIX error, and `#f` otherwise.

This function is defined in **SRFI-170** (*POSIX API*).

STklos procedure

```
(posix-error-name posix-error)
```

This procedure returns a symbol that is the name associated with the value of `errno` when the POSIX function reported an error. This can be used to provide programmatic recovery when a POSIX function can return more than one value of `errno`.

This function is defined in **SRFI-170** (*POSIX API*).

STklos procedure

```
(posix-error-message posix-error)
```

This procedure returns a string that is an error message reflecting the value of errno when the POSIX function reported an error. This string is useful for reporting the cause of the error to the user

This function is defined in **SRFI-170** (*POSIX API*).

STklos procedure

```
(posix-error-errno posix-error)
```

This procedure returns the value of `errno` (an exact integer).

STklos procedure

```
(posix-error-procedure posix-error)
```

This procedure returns the name of the Scheme procedure that raised the error.

STklos procedure

```
(posix-error-args posix-error)
```

This procedure returns the list of the Scheme procedure arguments that raised the error.

4.13.3. Environment

STklos procedure

```
(getenv str)
(getenv)
```

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") ...)
```

STklos procedure

```
(setenv! var value)
```

Sets the environment variable `var` to `value`. `Var` and `value` must be strings. The result of `setenv!` is ***void***.

STklos procedure

```
(unsetenv! var)
```

Unsets the environment variable `var`. `Var` must be a string. The result of `unsetenv!` is ***void***.

STklos defines also the R^{7}RS (and **SRFI-96**) standard primivitives to acess environment variables.

R^{7}RS procedure

```
(get-environment-variable name)
```

Returns the value of the named environment variable as a string, or `#f` if the named environment variable is not found. The name argument is expected to be a string. This function is similar to the `getenv`. It has been added to be support **SRFI-98** (*Interface to access environment variables*).

R^{7}RS procedure

```
(get-environment-variables)
```

Returns names and values of all the environment variables as an a-list. This function is defined by

SRFI-98 (*Interface to access environment variables*).

STklos procedure

```
(build-path-from-shell-variable var)
(build-path-from-shell-variable var sep)
```

Builds a path as a list of strings (which is the way STklos represents paths) from the environment variable `var`, given the separator characters given in `sep` (which defaults to `":"`, the standrad Unix path separator). If the `var` is not definied in the environment, `build-path-from-shell-variable` returns the empty list.

If the shell variable `MYPATH` is "/bin:/sbin:/usr/bin"`, then

```
(build-path-from-shell-variable "MYPATH")       => ("/bin" "/sbin" "/usr/bin")
(build-path-from-shell-variable "MYPATH" "/:")  => ("bin" "sbin" "usr" "bin")
```

4.13.4. Time

R^{7}RS procedure

```
(current-second)
```

Returns an inexact number representing the current time on the International Atomic Time (TAI) scale. The value 0.0 represents midnight on January 1, 1970 TAI (equivalent to ten seconds before midnight Universal Time) and the value 1.0 represents one TAI second later.

R^{7}RS procedure

```
(current-jiffy)
```

Returns the number of *jiffies* as an exact integer that have elapsed since an arbitrary, implementation-defined epoch. A jiffy is an implementation-defined fraction of a second which is defined by the return value of the `jiffies-per-second` procedure. The starting epoch is guaranteed to be constant during a run of the program, but may vary between runs.

R^{7}RS procedure

```
(jiffies-per-seconds)
```

Returns an exact integer representing the number of jiffies per second.

```
(define (time-length)
  (let ((list (make-list 100000))
        (start (current-jiffy)))
    (length list)
    (/ (- (current-jiffy) start)
       (jiffies-per-second))))
```

STklos procedure

```
(clock)
```

Returns an approximation of processor time, in milliseconds, used so far by the program.

STklos procedure

```
(sleep n)
```

Suspend the execution of the program for at `ms` milliseconds. Note that due to system clock resolution, the pause may be a little bit longer. If a signal arrives during the pause, the execution may be resumed.

STklos syntax

```
(time expr1 expr2 …)
```

Evaluates the expressions `expr1`, `expr2`, ... and returns the result of the last expression. This form prints also the time spent for this evaluation, in milliseconds, on the current error port. This is CPU time, and not real ("wall") time.

4.13.5. System Information

R^{7}RS procedure

```
(features)
```

Returns a list of the feature identifiers which `cond-expand` treats as true. Here is an example of what `features` might return:

```
(features) => (STklos STklos-0.3 exact-complex
               ieee-float full-unicode ratios little-endian ...)
```

STklos procedure

```
(running-os)
```

Returns the name of the underlying Operating System which is running the program. The value returned by `runnin-os` is a symbol. For now, this procedure returns either `unix`, `android`, `windows`, or `cygwin-windows`.

STklos procedure

```
(hostname)
```

Return the host name of the current processor as a string.

R^{7}RS procedure

```
(command-line)
```

Returns the command line passed to the process as a list of strings. The first string corresponds to the command name.

STklos procedure

```
(command-name)
```

Returnd the name of the running program if it is a standalone and #f otherwise. This function is defined in **SRFI-193** (*Command line*).

STklos procedure

```
(command-args)
(argv)
```

Returns a list of the arguments given on the shell command line. The interpreter options are no included in the result. The name `argv` is deprecated and should not be used.

STklos procedure

```
(argc)
```

Returns the number of arguments present on the command line.

STklos procedure

```
(program-name)
```

Returns the invocation name of the current program as a string. If the file is not a script (in sense of **SRFI-193**), it is the name of the running ***STklos*** interpreter, otherwise it is the name of the running script. This function always returns a string whereas the `command-name` procedure returns `#f` when the program name is not a script.

STklos procedure

```
(script-file)
```

Returns the absolute path of the current script. If the calling program is not a script, #f is returned. This function is defined in **SRFI-193** (*Command line*).

STklos procedure

```
(script-directory)
```

Returns the non-filename part of script-file as a string. As with `script-file`, this is an absolute pathname.

STklos procedure

```
(version)
(implementation-version)
```

Returns a string identifying the current version of the system. A version is constituted of two numbers separated by a point: the version and the release numbers. Note that `implementation-version` corresponds to the **SRFI-112** (*Environment Inquiry*) name of this function.

STklos procedure

```
(machine-type)
```

Returns a string identifying the kind of machine which is running the program. The result string is of the form `[os-name]-[os-version]-[cpu-architecture]`.

STklos procedure

```
(implementation-name)
```

This function is defined in **SRFI-112** (*Environment Inquiry*); it returns the Scheme implementation (i.e. the string `"STklos"`).

STklos procedure

```
(cpu-architecture)
```

This function is defined in **SRFI-112** (*Environment Inquiry*); it returns the CPU architecture, real or virtual, on which this implementation is executing.

STklos procedure

```
(machine-name)
```

This function is defined in **SRFI-112** (*Environment Inquiry*); it returns a name for the particular machine on which the implementation is running.

STklos procedure

```
(os-name)
```

This function is defined in **SRFI-112** (*Environment Inquiry*); it returns the name for the operating system, platform, or equivalent on which the implementation is running.

STklos procedure

```
(os-version)
```

This function is defined in **SRFI-112** (*Environment Inquiry*); it returns the version for the operating system, platform, or equivalent on which the implementation is running.

STklos procedure

```
(getpid)
```

Returns the system process number of the current program (i.e. the Unix *PID* as an integer).

4.13.6. 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** (*Running Scheme Scripts on Unix*) on how to use a `main` function in a Scheme program.

STklos procedure

```
(parse-arguments <args> <clause1> <clause2> …)
```

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

(define (main args)
  (parse-arguments args
     "Usage: foo [options] [parameter ...]"
     "General options:"
         (("verbose" :alternate "v" :help "be more verbose")
           (printf "Seen the verbose option~%"))
         (("long" :help "a long option alone")
           (printf "Seen the long option~%"))
         (("s" :help "a short option alone")
           (printf "Seen the short option~%"))
     "File options:"
         (("input" :alternate "f" :arg file
                   :help "use <file> as input")
           (printf "Seen the input option with ~S argument~%" file))
         (("output" :alternate "o" :arg file
                    :help "use <file> as output")
           (printf "Seen the output option with ~S argument~%" file))
     "Misc:"
         (("help" :alternate "h"
                  :help "provides help for the command")
           (arg-usage (current-error-port))
           (exit 1))
     (else
         (printf "All options parsed. Remaining arguments are ~S~%"
                   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
```

Notes:

- 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`, or
  - `-o bar`, or
  - `-obar`

STklos procedure

```
(arg-usage port)
(arg-usage port as-sexpr)
```

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 *Sexpr*s. This is useful if the help strings need to be manipulated by a program.

4.13.7. Misc. System Procedures

STklos procedure

```
(system string)
```

Sends the given `string` to the system shell `/bin/sh`. The result of `system` is the integer status code the shell returns.

STklos procedure

```
(exec str)
(exec-list str)
```

These procedures execute the command given in `str`. The command given in `str` is passed to `/bin/sh`. `Exec` returns a string 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")
```

STklos procedure

```
(address-of obj)
```

Returns the address of the object `obj` as an integer.

STklos procedure

```
(exit)
(exit ret-code)
```

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).

> The ***STklos*** `exit` primitive accepts also an integer value as parameter (R^{7}RS accepts only a boolean).

STklos procedure

```
(emergency-exit)
(emergency-exit ret-code)
```

Terminates the program without running any outstanding dynamic-wind *after* procedures and communicates an exit value to the operating system in the same manner as `exit`.

> The ***STklos*** `emergency-exit` primitive accepts also an integer value as parameter (R ^{7}RS accepts only a boolean).

STklos procedure

```
(die message)
(die message status)
```

`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.

STklos procedure

```
(get-password)
```

This primitive permits to enter a password (character echoing being turned off). The value returned by `get-password` is the entered password as a string.

STklos procedure

```
(register-exit-function! proc)
```

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 (or 0 if the programe terminates normally). 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)
                               (delete-file tmp))))
  out)
```

4.14. Keywords

Keywords are symbolic constants which evaluate to themselves. By default, a keyword is a symbol whose first (or last) character is a colon (*":"*). Alternatively, to be compatible with other Scheme implementations, the notation `#:foo` is also available to denote the keyword of name `foo`.

Note that the four directives `keyword-colon-position-xxx` or the parameter object ` keyword-colon-position` permit to change the default behavior. See section~Identifiers for more information.

STklos procedure

```
(keyword obj)
```

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

```
(keyword? 'foo)     => #f
(keyword? ':foo)    => #t  ; depends of keyword-colon-position
(keyword? 'foo:)    => #t  ; depends of keyword-colon-position
(keyword? '#:foo)   => #t  ; always
(keyword? :foo)     => #t  ; depends of keyword-colon-position
(keyword? foo:)     => #t  ; depends of keyword-colon-position
```

```
(keyword? #:foo)     => #t   ; always
```

STklos procedure

```
(make-keyword s)
```

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
```

STklos procedure

```
(keyword→string key)
```

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

STklos procedure

```
(string→keyword str)
```

This function function has been added to be compatibe with SRFI-88. It is equivalent to make-keyword, except that the parameter cannot be a symbol.

STklos procedure

```
(key-get list key)
(key-get list key default)
```

`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
```

STklos procedure

```
(key-set! list key value)
```

`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 ,(emph "physically") changed.

```
(let ((l (list #:one 1 #:two 2)))
  (set! l (key-set! l #:three 3))
  (cons (key-get l #:one)
        (key-get l #:three)))            => (1 . 3)
```

STklos procedure

```
(key-delete list key)
(key-delete! list key)
```

`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 as `key-delete` by physically modifying its `list` argument.

```
(key-delete '(:one 1 :two 2) :two)    => (:one 1)
(key-delete '(:one 1 :two 2) :three)  => (:one 1 :two 2)
(let ((l (list :one 1 :two 2)))
   (key-delete! l :two)
   l)                                 =>  (:one 1)
```

STklos procedure

```
(keyword-colon-position)
(keyword-colon-position value)
```

This parameter object indicates the convention used by the reader to denote keywords. The allowed values are:

- **none**, to forbid a symbol with colon to be interpreted as a keyword,
- **before**, to read symbols starting with a colon as keywords,

- **after**, to read symbols ending with a colon as keywords,
- **both**, to read symbols starting or ending with a colon as keywords.

Note that the notation `#:key` is always read as a keyword independently of the value of `keyword-colon-position`. Hence, we have

```
(list (keyword? ':a)
      (keyword? 'a:)
      (keyword? '#:a))
                 => (#f #f #t)  ; if keyword-colon-position is none
                 => (#t #f #t)  ; if keyword-colon-position is before
                 => (#f #t #t)  ; if keyword-colon-position is after
                 => (#t #t #t)  ; if keyword-colon-position is both
```

4.15. 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.

STklos hash tables procedures are identical to the ones defined in **SRFI-69** (*Basic Hash Tables*). Note that the default comparison function is `eq?` whereas it is `equal?` in this SRFI. See SRFI's documentation for more information.

STklos procedure

```
(make-hash-table)
(make-hash-table comparison)
(make-hash-table comparison hash)
```

`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. Objets 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 `hash-table-hash` *primitive*).
- `comparison` defaults to the `eq?` procedure (see `eq?` *primitive*)).

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)))))))))
```

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

STklos procedure

```
(hash-table? obj)
```

Returns `#t` if `obj` is a hash table, returns `#f` otherwise.

STklos procedure

```
(hash-table-hash obj)
```

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 Common Lisp `sxhash` function from which this procedure is modeled.

STklos procedure

```
(alist→hash-table alist)
(alist→hash-table alist comparison)
(alist→hash-table alist comparison hash)
```

Returns hash-table built from the *association list* `alist`. This function maps the `car` of every element in `alist` to the `cdr` of corresponding elements in `alist`. the `comparison` and `hash` functions are interpreted as in `make-hash-table`. If some key occurs multiple times in `alist`, the value in the first association will take precedence over later ones.

STklos procedure

```
(hash-table→alist hash)
```

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.

the order of pairs in the resulting list is unspecified.

```
(let ((h (make-hash-table)))
  (dotimes (i 5)
    (hash-table-set! h i (number->string i)))
  (hash-table->alist h))
      => ((3 . "3") (4 . "4") (0 . "0")
          (1 . "1") (2 . "2"))
```

STklos procedure

```
(hash-table-set! hash key value)
```

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

STklos procedure

```
(hash-table-ref hash key)
(hash-table-ref hash key thunk)
```

Returns the value associated with `key` in the given `hash` table. If no value has been associated with `key` in `hash`, the specified `thunk` is called and its value is returned; otherwise an error is raised.

```
(define h1 (make-hash-table))
(hash-table-set! h1 'foo (list 1 2 3))
(hash-table-ref  h1 'foo)                 =>  (1 2 3)
(hash-table-ref  h1 'bar
                    (lambda () 'absent))  =>  absent
(hash-table-ref  h1 'bar)                 =>  error
(hash-table-set! h1 '(a b c) 'present)
(hash-table-ref  h1 '(a b c)
                     (lambda () 'absent)) => absent

(define h2 (make-hash-table equal?))
(hash-table-set! h2 '(a b c) 'present)
(hash-table-ref  h2 '(a b c))             => present
```

STklos procedure

```
(hash-table-ref/default hash key default)
```

This function is equivalent to

```
(hash-table-ref hash key (lambda () default))
```

STklos procedure

```
(hash-table-delete! hash key)
```

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

```
(define h (make-hash-table))
(hash-table-set! h 'foo (list 1 2 3))
(hash-table-ref h 'foo)                 => (1 2 3)
(hash-table-delete! h 'foo)
(hash-table-ref h 'foo
                 (lambda () 'absent)  => absent
```

STklos procedure

```
(hash-table-exists? hash key)
```

Returns #t if there is any association of key in hash. Returns #f otherwise.

STklos procedure

```
(hash-table-update! hash key update-fun thunk)
(hash-table-update!/default hash key update-fun default)
```

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 first inserted before updating it (this new entry being the result of calling thunk).

Note that the expression

```
(hash-table-update!/default hash key update-fun default)
```

is equivalent to

```
(hash-table-update! hash key update-fun (lambda () default))
```

```
(let ((h   (make-hash-table))
      (1+  (lambda (n) (+ n 1))))
  (hash-table-update!/default h 'test 1+ 100)
  (hash-table-update!/default h 'test 1+)
  (hash-table-ref h 'test))             => 102
```

STklos procedure

```
(hash-table-for-each hash proc)
(hash-table-walk hash proc)
```

`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***.

> The order of application of `proc` is unspecified.

> `hash-table-walk` is another name for `hash-table-for-each` (this is the name used in **SRFI-69** (*Basic Hash Tables*).

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

STklos procedure

```
(hash-table-map hash proc)
```

`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.

> The order of application of `proc` is unspecified.

```
(let ((h (make-hash-table)))
  (dotimes (i 5)
    (hash-table-set! 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"))
```

STklos procedure

```
(hash-table-keys hash)
```

```
(hash-table-values hash)
```

Returns the keys or the values of `hash`.

STklos procedure

```
(hash-table-fold hash func init-value)
```

This procedure calls `func` for every association in `hash` with three arguments: the key of the association key, the value of the association value, and an accumulated value, `val`. `Val` is init-value for the first invocation of `func`, and for subsequent invocations of `func`, the return value of the previous invocation of `func`. The value `final-value` returned by `hash-table-fold` is the return value of the last invocation of `func`. The order in which `func` is called for different associations is unspecified.

For instance, the following expression

```
(hash-table-fold ht (lambda (k v acc) (+ acc 1)) 0)
```

computes the number of associations present in the `ht` hash table.

STklos procedure

```
(hash-table-copy hash)
```

Returns a copy of `hash`.

STklos procedure

```
(hash-table-merge! hash1 hash2)
```

Adds all mappings in `hash2` into `hash1` and returns the resulting hash table. This function may modify `hash1` destructively.

STklos procedure

```
(hash-table-equivalence-function hash)
```

Returns the equivalence predicate used for keys in `hash`.

STklos procedure

```
(hash-table-hash-function hash)
```

Returns the hash function used for keys in `hash`.

STklos procedure

```
(hash-table-size hash)
```

Returns the number of entries in the `hash`.

STklos procedure

```
(hash-table-stats hash)
(hash-table-stats hash port)
```

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 `current-output-port` *primitive*).

4.16. Dates and Times

STklos stores dates and times with a compact representation which consists is an integer which represents the number of seconds elapsed since the ***Epoch*** (00:00:00 on January 1, 1970, Coordinated Universal Time --UTC). Dates can also be represented with date structures.

R^{7}RS procedure

```
(current-second)
```

Returns an inexact number representing the current time on the International Atomic Time (TAI) scale. The value 0.0 represents midnight on January 1, 1970 TAI (equivalent to ten seconds before midnight Universal Time) and the value 1.0 represents one TAI second later.

STklos procedure

```
(current-seconds)
```

Returns the time since the Epoch (that is 00:00:00 UTC, January 1, 1970), measured in seconds in the Coordinated Universal Time (UTC) scale.

This ***STklos*** function should not be confused with the R^{7}RS primitive `current-second` which returns an inexact number and whose result is expressed using the International Atomic Time instead of UTC.

STklos procedure

```
(current-time)
(current-time type)
```

Return the current time as time object. The type can be `time-utc` or `time-tai`. If omitted, `type` is `time-utc`.

To use more time types, such as `time-monotonic` and `time-process`, please load **SRFI-19**.

STklos procedure

```
(make-time nanosecond second)
(make-time type nanosecond second)
```

Creates a time structure with the given `nanosecond` and `second`. If `type` is passed, it must be a symbol representing one of the supported time types (`time-tai`, `time-utc`, `time-monotonic`, `time-process` and `time-duration`).

`time-monotonic`, `time-process` and `time-duration` can be created, but operations on them are only available when SRFI-19 is loaded.

STklos procedure

```
(time-type t)
(set-time-type! t v)
(time-second t)
(set-time-second! t s)
(time-nanosecond t)
```

```
(set-time-nanosecond! t n)
```

These are accessors for time structures.

STklos procedure

```
(time? obj)
```

Return `#t` if `obj` is a time object, othererwise returns `#f`.

STklos procedure

```
(time→seconds time)
```

Convert the time object `time` into an inexact real number representing the number of seconds elapsed since the Epoch.

```
(time->seconds (current-time))  ==>  1138983411.09337
```

STklos procedure

```
(seconds→time x)
```

Converts into a time object the real number `x` representing the number of seconds elapsed since the Epoch.

```
(seconds->time (+ 10 (time->seconds (current-time))))
         ==>  a time object representing 10 seconds in the future
```

STklos procedure

```
(time-utc→time-tai t)
(time-utc→time-tai! t)
```

Converts `t`, which must be of type `time-utc`, to the type `time-tai`.

`Time-utc→time-tai` creates a new object, while `time-utc→time-tai` can use `t` to build the returned object.

STklos procedure

```
(time-tai→time-utc t)
(time-tai→time-utc! t)
```

Converts `t`, which must be of type `time-tai`, to the type `time-utc`.

`Time-tai→time-utc` creates a new object, while `time-tai→time-utc` can use `t` to build the returned object.

STklos procedure

```
(current-date)
```

Returns the current system date.

STklos procedure

```
(make-date :key nanosecond second minute hour day month year zone-offset)
(make-date :optional nanosecond second minute hour day month year zone-offset)
```

Build a date from its argument. `hour`, `minute`, `second`, `nanosecond` default to 0; `day` and `month` default to 1; `year` defaults to 1970.

STklos procedure

```
(date? obj)
```

Return `#t` if `obj` is a date, and otherwise returns `#f`.

STklos procedure

```
(date-nanosecond d)
```

Return the nanosecond of date `d`.

STklos procedure

```
(date-second d)
```

Return the second of date `d`, in the range 0 to 59.

STklos procedure

```
(date-minute d)
```

Return the minute of date `d`, in the range 0 to 59.

STklos procedure

```
(date-hour d)
```

Return the hour of date `d`, in the range 0 to 23.

STklos procedure

```
(date-day d)
```

Return the day of date `d`, in the range 1 to 31

STklos procedure

```
(date-month d)
```

Return the month of date `d`, in the range 1 to 12

STklos procedure

```
(date-year d)
```

Return the year of date d.

STklos procedure

```
(date-week-day d)
```

Return the week day of date d, in the range 0 to 6 (0 is Sunday).

STklos procedure

```
(date-year-day d)
```

Return the the number of days since January 1 of date d, in the range 1 to 366.

STklos procedure

```
(date-dst d)
```

Return an indication about daylight saving adjustment of date d:

- 0 if no daylight saving adjustment
- 1 if daylight saving adjustment
- -1 if the information is not available

STklos procedure

```
(date-tz d)
```

Return the time zone of date d.

STklos procedure

```
(local-timezone-offset)
```

Returns the local timezone offset, in seconds.

For example, for GMT+2 it will be `2 * 60 * 60` = `7200`

```
(local-timezone-offset) => 0          ;; for GMT
(local-timezone-offset) => 7200       ;; for GMT+2
(local-timezone-offset) => -10800     ;; for GMT-3
```

The timezone is searched for in the environment variable `TZ`. If this variable does not appear in the environment, the system timezone is used.

STklos procedure

```
(date→seconds d)
```

Convert the date `d` to the number of seconds since the *Epoch*, 1970-01-01 00:00:00 +0000 (UTC).

```
(date->seconds (make-date 0 37 53 1 26 10 2012 0))   => 1351216417.0
```

STklos procedure

```
(date→string d)
(date→string d format)
```

Convert the date `d` using the string `format` as a specification. Conventions for format are the same as the one of *primitive* `seconds→string`. If `format` is omitted, it defaults to `"~c"`.

STklos procedure

```
(seconds→date n)
```

Convert the date `n` expressed as a number of seconds since the *Epoch*, 1970-01-01 00:00:00 +0000 (UTC) into a date. `n` can be an exact integer or an inexact real.

This is equivalent to converting time-UTC to date.

```
(seconds->date 1351216417)           => #[date 2012-10-26 1:53:37]
```

STklos procedure

```
(seconds→string format n)
```

Convert a date expressed in seconds using the string `format` as a specification. Conventions for `format` are given below:

- ~~ a literal ~
- **~a** locale's abbreviated weekday name (Sun...Sat)
- **~A** locale's full weekday name (Sunday...Saturday)
- **~b** locale's abbreviate month name (Jan...Dec)
- **~B** locale's full month day (January...December)
- **~c** locale's date and time (e.g., ,(code "Fri Jul 14 20:28:42-0400 2000"))
- **~d** day of month, zero padded (01...31)
- **~D** date (mm/dd/yy)
- **~e** day of month, blank padded (1...31)
- **~f** seconds+fractional seconds, using locale's decimal separator (e.g. 5.2).
- **~h** same as ~b
- **~H** hour, zero padded, 24-hour clock (00...23)
- **~I** hour, zero padded, 12-hour clock (01...12)
- **~j** day of year, zero padded
- **~k** hour, blank padded, 24-hour clock (00...23)
- **~l** hour, blank padded, 12-hour clock (01...12)
- **~m** month, zero padded (01...12)
- **~M** minute, zero padded (00...59)
- **~n** new line
- **~p** locale's AM or PM
- **~r** time, 12 hour clock, same as "~I:~M:~S ~p"
- **~s** number of full seconds since *the epoch* (in UTC)
- **~S** second, zero padded (00...61)
- **~t** horizontal tab
- **~T** time, 24 hour clock, same as "~H:~M:~S"
- **~U** week number of year with Sunday as first day of week (00...53)
- **~V** weekISO 8601:1988 week number of year (01...53) (week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week)
- **~w** day of week (1...7, 1 is Monday)
- **~W** week number of year with Monday as first day of week (01...52)
- **~x** week number of year with Monday as first day of week (00...53)

- **~X** locale’s date representation, for example: "07/31/00"
- **~y** last two digits of year (00…99)
- **~Y** year
- **~z** time zone in RFC-822 style
- **~Z** symbol time zone

STklos procedure

```
(seconds→list sec)
```

Returns a keyword list for the date given by `sec` (a date based on the Epoch). The keyed values returned are

- nanosecond : 0 to 999999
- 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 : indication about daylight savings time (see *primitive* `date-dst`).
- tz : the difference between Coordinated Universal Time (UTC) and local standard time in seconds.])

```
(seconds->list (current-second))
       => (#:nanosecond 182726 #:second 21 #:minute 35 #:hour 20 #:day 10 #:month 1
           #:year 2022 #:week-day 1 #:year-day 10 #:dst 0 #:tz -3600)
```

STklos procedure

```
(date)
```

Returns the current date in a string.

4.17. Boxes

Boxes are objects which contain one or several states. A box may be constructed with the box, constant-box. ***STklos*** boxes are compatible with the one defined in **SRFI-111** (*Boxes*) or **SRFI-195** (*Multiple-value boxes*). Boxes of SRFI-111 can contain only one value, whereas SRFI-195 boxes can contain multiple values. Furthermore, ***STklos*** defines also the notion of constant boxes which are not mutable.

The read primitive can also make single valued boxes (using the `#&` notation). Such boxes are mutable.

Note that two boxes are `equal?` ***iff*** their content are `equal?`.

STklos procedure

```
(box value …)
(make-box value …)
```

Returns a new box that contains all the given `value`s. The box is mutable.

```
(let ((x (box 10)))
  (list 10 x))          => (10 #&10)
```

The name `make-box` is now obsolete and kept only for compatibility.

STklos procedure

```
(constant-box value …)
(make-constant-box value …)
```

Returns a new box that contains all the given `value`s. The box is immutable.

The name `make-constant-box` is now obsolete and kept only for compatibility.

STklos procedure

```
(box? obj)
```

Returns `#t` if `obj` is a box, `#f` otherwise.

STklos procedure

```
(box-mutable? obj)
```

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

STklos procedure

```
(set-box! box value …)
(box-set! box value …)
```

Changes `box` to hold `value`s. It is an error if `set-box!` is called with a number of values that differs from the number of values in the box being set. (In other words, `set-box!` does not allocate memory.) It is also an error to call `set-box!` on a box which is not mutable.

The name `box-set!` is now obsolete and kept only for compatibility.

STklos procedure

```
(unbox box)
```

Returns the values currently in `box`.

STklos procedure

```
(box-arity box)
```

Returns the number of values in `box`.

STklos procedure

```
(unbox-value box i)
```

Returns the `i`th value of `box`. It is an error if `i` is not an exact integer between 0 and `n`-1, when `n` is the number of values in `box`.

STklos procedure

```
(set-box-value! box i obj)
```

Changes the `i`th value of `box` to `obj`. It is an error if `i` is not an exact integer between 0 and `n`-1, when `n` is the number of values in `box`.

4.18. 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.

STklos procedure

```
(run-process command p1 p2 …)
```

`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 abolute 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")
```

STklos procedure

```
(process? obj)
```

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

STklos procedure

```
(process-alive? proc)
```

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

STklos procedure

```
(process-pid proc)
```

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

STklos procedure

```
(process-input proc)
(process-output proc)
(process-error proc)
```

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`.

STklos procedure

```
(process-wait proc)
```

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

STklos procedure

```
(process-exit-status proc)
```

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

STklos procedure

```
(process-send-signal proc sig)
```

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 Section 4.20). The result of `process-send-signal` is ***void***.

STklos procedure

```
(process-kill proc)
```

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

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

STklos procedure

```
(process-stop proc)
(process-continue proc)
```

`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)
```

STklos procedure

```
(process-list)
```

Returns the list of processes which are currently running (i.e. alive).

STklos procedure

```
(fork)
(fork thunk)
```

This procedure is a wrapper around the standard Unix `fork` system call which permits to create a new (heavy) process. When called without parameter, this procedure returns two times (one time in the parent process and one time in the child process). The value returned to the parent process is a process object representing the child process and the value returned to the child process is always the value `#f`. When called with a parameter (which must be a thunk), the new process excutes `thunk` and terminate it execution when `thunk` returns. The value returned to the parent process is a process object representing the child process.

4.19. 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.

STklos procedure

```
(make-client-socket hostname port-number)
(make-client-socket hostname port_number line-buffered)
```

`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 tranmitted as soon as a `"#\newline` character is encountered). The default value of `line-buffered` is `#t`.

STklos procedure

```
(make-server-socket)
(make-server-socket port-number)
```

`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.

STklos procedure

```
(socket-shutdown sock)
(socket-shutdown sock close)
```

`Socket-shutdown` shutdowns the connection associated to `socket`. If the socket is a server socket, `socket-shutdown` is called on all the client sockets connected to this server. `Close` indicates if 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))))
```

STklos procedure

```
(socket-accept socket)
(socket-accept socket line-buffered)
```

`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 12345 [1]

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 12345))
      (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`.

STklos procedure

```
(socket? obj)
```

Returns `#t` if `socket` is a socket, otherwise returns `#f`.

STklos procedure

```
(socket-server? obj)
```

Returns `#t` if `socket` is a server socket, otherwise returns `#f`.

STklos procedure

```
(socket-client? obj)
```

Returns `#t` if `socket` is a client socket, otherwise returns `#f`.

STklos procedure

```
(socket-host-name socket)
```

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`.

STklos procedure

```
(socket-host-address socket)
```

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`.

STklos procedure

```
(socket-local-address socket)
```

Returns a string which contains the IP number of the local host attached to `socket`.

STklos procedure

```
(socket-port-number socket)
```

Returns the integer number of the port used for `socket`.

STklos procedure

```
(socket-input socket)
(socket-output socket)
```

Returns the port associated for reading or writing with the program connected with `socket`. Note that this port is both textual and binary. If no connection has already been established, these functions return `#f`.

The following example shows how to make a client socket. Here we create a socket on port 13 of the machine `kaolin.unice.fr` [2]:

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (format #t "Time is: ~A~%" (read-line (socket-input s)))
  (socket-shutdown  s))
```

4.20. Signals

STklos permits to associate handlers to POSIX.1 signals. When a signal handler is called, the integer value of this signal is passed to it as (the only) parameter.

The following POXIX.1 values for signal numbers are defined: `SIGABRT` `SIGALRM`, `SIGFPE`, `SIGHUP`,`SIGILL`, `SIGINT`, `SIGKILL`, `SIGPIPE`, `SIGQUIT`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGCHLD`, `SIGCONT`, `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`. Moreover, the following constants, which are often available on most systems are also defined (if supported by the running system): `SIGTRAP`, `SIGIOT`, `SIGEMT`, `SIGBUS`, `SIGSYS`, `SIGURG`, `SIGCLD`, `SIGIO`, `SIGPOLL`, `SIGXCPU`, `SIGXFSZ`, `SIGVTALRM`, `SIGPROF`, `SIGWINCH`, `SIGLOST`.

See your Unix documentation for the exact meaning of each constant or [POSIX]. Use symbolic constants rather than their numeric value if you plan to port your program on another system.

STklos procedure

```
(set-signal-handler! sig handler)
```

Replace the handler for integer signal `sig` with `handler`. The value of `handler` can be:

- `#t` to reset the signal handler for `sig` to the default system handler.
- `#f` to ignore the `sig` signal. Note that POSIX states that `SIGKILL` and `SIGSTOP` cannot be ignored or caught.
- a one parameter procedure, which will be called when the processus receives the signal `sig`.

This procedure returns ***void***.

```
(let ((x #f))
  (set-signal-handler! SIGUSR1
                       (lambda (i) (set! x #t)))
  (send-signal SIGUSR1)
  x)    => #t
```

STklos procedure

```
(get-signal-handler! sig)
```

Return the handler for integer signal `sig`. The value of `handler` can be a boolean value or a procedure. See primitive `set-signal-handler!` for more information.

STklos procedure

```
(send-signal sig)
```

```
(send-signal sig pid)
```

Send the integer signal `sig` to the process with `pid` process id. If the second parameter is absent, it deaults to the one of the running program.

4.21. Parameter Objects

STklos parameters correspond to the ones defined in **SRFI-39** (*Parameters objects*). See SRFI document for more information.

STklos procedure

```
(make-parameter init)
(make-parameter init converter)
```

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 'write-shared "bad boolean ~S" x)))))

 (radix)           =>  10
 (radix 2)
 (radix)           =>  2
 (write-shared 0)  => error

 (define prompt
   (make-parameter
     123
     (lambda (x)
       (if (string? x)
           x
           (with-output-to-string (lambda () (write x)))))))

 (prompt)        =>  "123"
 (prompt ">")
 (prompt)        =>  ">"
```

STklos syntax

```
(parameterize ((expr1 expr2) ...) <body>)
```

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^{5}RS 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"
```

STklos procedure

```
(parameter? obj)
```

Returns `#t` if `obj` is a parameter object, otherwise returns `#f`.

4.22. Misc

STklos procedure

```
(gc)
```

Force a garbage collection step.

STklos procedure

```
(void)
(void arg1 …)
```

Returns the special ***void*** object. If arguments are passed to `void`, they are evalued and simply ignored.

STklos procedure

```
(void? obj)
```

Returns `#t` is `obj` is `#void`, and `#f` otherwise. The usual "unspecified" result in Scheme standard and in SRFIs is `#void` in STklos, and it is also returned by the procedure `void`.

```
(void? (void))                 => #t
(define x (if #f 'nope))
(void? x)                      => #t
(void? '())                    => #f
(void? 'something)             => #f
(void? (for-each print '(1 2 3))) => #t
```

STklos procedure

```
(error str obj …)
(error name str obj …)
```

`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.

The specification string may follow the *tilde conventions* of `format` (see *primitive* `format`); in this case this procedure builds an error message according to the specification given in `str`. Otherwise, this procedure is in conformance with the `error` procedure defined in **SRFI-23** (*Error reporting mechanism*) and `str` is printed with the `display` procedure, whereas the `obj`s 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
```

STklos procedure

```
(signal-error cond str obj …)
(signal-error cond name str obj …)
```

This procedure is similar to error, except that the type of the error can be passed as the first parameter. The type of the error must be a condition which inherits from `&error-message`.

Note that `(error arg …)` is equivalent to

```
(signal-error &error-message arg ...)
```

R^{7}RS procedure

```
(read-error? obj)
(file-error? obj)
```

Error type predicates. Returns #t if `obj` is an object raised by the read procedure or by the inability to open an input or output port on a file, respectively. Otherwise, it returns #f.

R^{7}RS procedure

```
(error-object? obj )
```

Returns #t if obj is an object created by error. Otherwise, it returns #f.

R^{7}RS procedure

```
(error-object-message error-object)
```

Returns the message encapsulated by `error-object`.

R^{7}RS procedure

```
(error-object-irritants error-object)
```

Returns the message encapsulated by `error-object`.

STklos procedure

```
(error-object-location error-object)
```

Returns the location encapsulated by `error-object` if it exists. Returns #f otherwise. The location corresponds generally to the name of the procedure which raised the error.

```
(guard (cnd
         (else (error-object-location cnd)))
  (error 'foo "error message"))  => foo
```

STklos syntax

```
(require-extension <clause> …)
```

The syntax of require-extension is as follows:

```
(require-extension <clause> ...)
```

A clause may have the form:

1. `(srfi number …)`
2. `(identifier …)`
3. `identifier`

In the first form the functionality of the indicated SRFIs are made available in the context in which the `require-extension` form appears. For instance,

```
(require-extension (srfi 1 2)) ; Make the SRFI 1 and 2 available
```

This form is compatible with **SRFI-55** (*Require-extension*).

The second and third forms are ***STklos*** extensions. If the form is a list, it is equivalent to an import. That is,

```
(require-extension (streams primitive) (streams derived))
```

is equivalent to

```
(import (streams primitive) (streams derived))
```

The final form permits to use symbolic names for requiring some extensions. For instance,

```
(require-extension lists and-let*)
```

is equivalent to the requiring `srfi-1` and `srfi-2`.

A list of available symbolic names for features is given in Chapter 14.

STklos procedure

```
(require-feature feature)
```

This primitive ensures that the `feature` (in sense of SRFI-0 feature) can be used. In particular, it eventually requires the loading of the files needed to used `feature`. The `feature` can be expressed as a string or a symbol, If feature is an integer `n`, it is equivalent to `srfi-n`. Consequently, to use **SRFI-1** the following forms are equivalent:

```
(require-feature 'srfi-1)
(require-feature "srfi-1")
(require-feature 1)
```

```
(require-feature 'lists)       ;; Since this feature name is an alias for SRFI-1
```

See also Chapter 14 for more information.

STklos procedure

```
(repl)
(repl :in inport :out outport :err errport)
```

This procedure launches a new Read-Eval-Print-Loop. Calls to `repl` can be embedded. The ports used for input/output as well as the error port can be passed when `repl` is called. If not passed, they default to `current-input-port`, `current-output-port` and `current-error-port`.

STklos syntax

```
(assume obj ...)
```

The special form `assume` is defined in **SRFI-145** (*Assumptions*). When ***STklos*** is in debug mode, this special form is an expression that evaluates to the value of `obj` if `obj` evaluates to a true value and it is an error if `obj` evaluates to a false value.

When ***STklos*** is not in debug mode, the call to `assume` is elided.

STklos procedure

```
(version-alist)
```

This function returns an association list of STklos properties as defined by **SRFI-176** (*Version flag*).

STklos procedure

```
(apropos obj)
(apropos obj module)
```

`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).

STklos procedure

```
(help obj)
(help)
```

When called with an argument, `help` tries to give some help on the given object, which could be a symbol, a procedure, a generic function or a method. Whe called called without arguments, `help` enters a read-help-print loop. The documentation for an object is searched in the object itself or, if absent, in STklos documentation. Inserting the documentation in an objet is very similar to Emacs docstrings: a documentation string is defined among the code. Exemples of such strings are given below

```
(define (foo n)
  "If the function body starts with a string, it's a docstring"
  (+ n 1))

(define-generic bar
  :documentation "Generic function docsting for bar")

(define-method bar ((x <integer>))
  "Probably less useful: as in functions, methods can have docstrings"
  (- x 1))
```

STklos procedure

```
(describe obj)
```

Shows a brief description of `obj`. If the object is structured such as a struct, class or instance, some information about its internal structure will be shown.

Using `describe` on simple values

```
(describe 5)
  5 is an integer.

(describe 5.4)
  5.4 is a real.

(describe 2+3i)
  2+3i is a complex number.

(describe #\A)
  #\A is a character, ascii value is 65.
```

Using describe on a class

```
(describe <integer>)
  <integer> is a class. It's an instance of <class>.
  Superclasses are:
      <rational>
  (No direct slot)
  (No direct subclass)
  Class Precedence List is:
      <integer>
      <rational>
      <real>
      <complex>
      <number>
      <top>
  (No direct method)
```

Using describe on structures

```
(define-struct person name email)
(define one (make-person "Some One" "one@domain.org"))

(describe person)
  #[struct-type person 139786494092352] is a structure type whose name is person.
  Parent structure type: #f
  Slots are:
       name
       email

(describe one)
  #[struct person 139786494288064] is an instance of the structure type person.
  Slots are:
       name = "Some One"
       email = "one@domain.org"
```

STklos syntax

```
(trace f-name …)
```

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.

STklos syntax

```
(untrace f-name …)
```

Invoking `untrace` with one or more function names causes the functions named not to be traced anymore.

Calling `untrace` with no argument will untrace all the functions currently traced.

STklos procedure

```
(pretty-print sexpr :key port width)
(pp sexpr :key port width)
```

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 if 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`.

STklos procedure

```
(procedure-formals proc)
```

Returns the formal parameters of procedure `proc`. Note that procedure formal parameters are kept in memory only if the compiler flag <<"compiler:keep-formals">> is set at its creation. If `proc` formal parameters are not available, `procedure-formals` returns `#f`.

STklos procedure

```
(procedure-source proc)
```

Returns the source form used to define procedure `proc`. Note that procedure source is kept in memory only if the compiler flag <<"compiler:keep-source">> is set at its creation. If `proc` source is not available, `procedure-source` returns `#f`.

STklos procedure

```
(ansi-color e1 e2 … en)
```

`ansi-color` permits to build a string which embeds ANSI codes to colorize texts on a terminal. Each expression e_i must be a string, a symbol or an integer.

Strings constitute the message to be displayed.

A symbol can designate

- a color in the set {`black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`} for foreground colors
- a color in the set {`bg-black`, `bg-red`, `bg-green`, `bg-yellow`, `bg-blue`, `bg-magenta`, `bg-cyan`, `bg-white`} for background colors.
- a qualifier such as `normal`, `bold`, `italic`, `underline`, `blink`, `reverse` or `no-bold`, `no-italic`, `no-underline`, `no-blink`, `no-reverse`.

Integer values can be used for terminals which are able to display 256 colors. If the number is positive, it is used as a foreground color. Otherwise, it is uses as a background color. Note that not all the terminals are able to use more than eight colors.

For instance,

```
(display (ansi-color "a word in "
                     'bold 'red "RED" 'normal
                     " and another in "
                     'reverse 'blue "BLUE" 'normal))
```

will display the words BLUE and RED in color.

STklos procedure

```
(disassemble proc)
(disassemble proc port)
```

This function prints on the given port (by default the current output port) the instructions of procedure `proc`. The printed code uses an *ad-hoc* instruction set that should be quite understandable.

```
(define (fact n)
  (if (< n 2)
      1
      (* n (fact (- n 1)))))
```

The call `(disassemble fact)` will produce the following output:

```
    000:  LOCAL-REF0-PUSH
    001:  SMALL-INT            2
    003:  JUMP-NUMGE           2  ;; ==> 007
    005:  IM-ONE
    006:  RETURN
    007:  LOCAL-REF0-PUSH
    008:  PREPARE-CALL
    009:  LOCAL-REF0
    010:  IN-SINT-ADD2         -1
    012:  PUSH-GREF-INVOKE     0 1
    015:  IN-MUL2
    016:  RETURN
```

The code of a procedure may be patched after the first execution of `proc` to optimize it.

If `proc` is an anonymous function, you can use the special notation `#pxxx` to disassemble it: *

```
(disassemble #p7fee1dd82f80)  ;; (address-of (lambda() 42))

      000:  SMALL-INT            42
      002:  RETURN
```

STklos procedure

```
(disassemble-expr sexpr)
(disassemble-expr sexpr show-consts)
(disassemble-expr sexpr show-consts port)
```

This function prints on the given `port` (by default the current output port) the instructions of the given `sexpr`. If `show-consts` is true, the table of the contants used in `sexpr` is also printed.

```
(disassemble-expr '(begin
                      (define x (+ y 10))
                      (cons x y))
                    #t)
```

will print:

```
000:  GLOBAL-REF           0
002:  IN-SINT-ADD2         10
004:  DEFINE-SYMBOL        1
006:  GLOBAL-REF-PUSH      1
008:  GLOBAL-REF           0
```

```
010:  IN-CONS
011:

Constants:
0: y
1: x
```

STklos procedure

```
(uri-parse str)
```

Parses the string `str` as an RFC-2396 URI and return a keyed list with the following components

- `scheme` : the scheme used as a string (defaults to `"file"`)
- `user`: the user information (generally expressed as `login:password`)
- `host` : the host as a string (defaults to "")
- `port` : the port as an integer (0 if no port specified)
- `path` : the path
- `query` : the qury 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 "https://stklos.net")
   => (:scheme "https" :user "" :host "stklos.net" :port 443
        :path "/" :query "" :fragment "")

(uri-parse "https://stklos.net:8080/a/file?x=1;y=2#end")
    => (:scheme "http" :user "" :host "stklos.net" :port 8080
        :path "/a/file" :query "x=1;y=2" :fragment "end")

(uri-parse "http://foo:secret@stklos.net:2000/a/file")
    => (:scheme "http" :user "foo:secret" :host "stklos.net"
        :port 2000  :path "/a/file" :query "" :fragment "")

(uri-parse "/a/file")
   => (:scheme "file" :user "" :host "" :port 0 :path "/a/file"
       :query "" :fragment "")

(uri-parse "")
   => (:scheme "file"  :user "" :host "" :port 0 :path ""
       :query "" :fragment "")
```

STklos procedure

```
(string→html str)
```

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 &lt;test&gt;"
```

STklos procedure

```
(md5sum obj)
```

Return a string contening the md5 sum of `obj`. The given parameter can be a string or an open input port.

STklos procedure

```
(md5sum-file str)
```

Return a string contening the md5 sum of the file whose name is `str`.

STklos procedure

```
(base64-encode in)
(base64-encode in out)
```

Encode in Base64 the characters from input port `in` to the output port `out`. If `out` is not specified, it defaults to the current output port.

```
(with-input-from-string "Hello"
  (lambda ()
    (with-output-to-string
      (lambda ()
        (base64-encode (current-input-port)))))) => "SGVsbG8="
```

STklos procedure

```
(base64-decode in)
(base64-decode in out)
```

Decode the Base64 characters from input port `in` to the output port `out`. If `out` is not specified, it defaults to the current output port.

```
(with-input-from-string "SGVsbG8="
  (lambda ()
    (with-output-to-string
      (lambda ()
        (base64-decode (current-input-port))))))  => "Hello"
```

STklos procedure

```
(base64-encode-string str)
```

Return a string contening the contents of `str` converted to Base64 encoded format.

STklos procedure

```
(base64-encode-string str)
```

Decode the contents of `str` expressed in Base64.

[1] Under Unix, you can simply connect to a listening socket with the `telnet` of `netcat` command. For the given example, this can be achieved with `netcat localhost 12345`

[2] Port 13, if open, can be used for testing: making a connection to it permits to know the distant system's idea of the time of day.

Chapter 5. Regular Expressions

STklos uses the Philip Hazel’s Perl-compatible Regular Expression (PCRE) library for implementing regexps [PCRE]. 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 description \footnote{The latest release of PCRE is available from http://www.pcre.org/.

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

```
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:

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

\	general escape character
^	negate the class, but only if the first character
-	indicates character range
[	POSIX character class (only if followed by POSIX syntax)
]	terminates the character class

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

5.1. 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 escaping action 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 backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write `\\`.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between `\Q` and `\E`. This is different from Perl in that `$` and `@` are handled as literals in `\Q`…`\E` sequences in PCRE, whereas in Perl, ``$` and `@` cause variable interpolation. Note the following examples:

Pattern	**PCRE matches**	**Perl matches**
\Qabc$xyz\E	abc$xyz	abc followed by the contents of $xyz
\Qabc\$xyz\E	abc\$xyz	abc\$xyz
\Qabc\E\$\Qxyz\E	abc$xyz	abc$xyz

The `\Q⋯\E` sequence is recognized both inside and outside character classes.

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a	alarm, that is, the BEL character (hex 07)
\cx	*control-x*, where x is any character
\e	escape (hex 1B)
\f	formfeed (hex 0C)

\n	newline (hex 0A)
\r	carriage return (hex 0D)
\t	tab (hex 09)
\ddd	character with octal code ddd, or backreference
\xhh	character with hex code hh

The precise effect of `\x` is as follows: if `x` is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex 1A, but `\c{` becomes hex 3B, while `\c;` becomes hex 7B.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

The third use of backslash is for specifying generic character types:

\d	any decimal digit
\D	any character that is not a decimal digit
\s	any whitespace character
\S	any character that is not a whitespace character
\w	any *word* character
\W	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.

For compatibility with Perl, `\s` does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The `\s` characters are HT (9), LF (10), FF (12), CR (13), and space (32).

A *word* character is any letter or digit or the underscore character, that is, any character which can be part of a Perl *word*. The definition of letters and digits is controlled by PCRE’s character tables, and may vary if locale-specific matching is taking place. For example, in the "fr" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by `\w`.

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.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

\b	matches at a word boundary
\B	matches when not at a word boundary
\A	matches at start of subject
\Z	matches at end of subject or before newline at end
\z	matches at end of subject
\G	matches at first matching position in subject

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode.

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 “\\”.

5.2. 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).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

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 need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. 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 newline 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.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A` it is always anchored, whether *multiline* is set or not.

5.3. Full Stop (period, 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.

5.4. Square Brackets

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. A matched character must be in the set of characters defined by the class, unless the first character in the class definition 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. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

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 caseful 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.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as

`[W-]46]` is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so `(W\]46)` is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example `[\000-\037]`.

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, and if character tables for the "fr" locale are in use, `[\xc8-\xcb]` matches accented E characters in both cases.

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.

All non-alphameric characters other than `\`, `-`, `^` (at the start) and the terminating `]` are non-special in character classes, but it does no harm if they are escaped.

5.5. POSIX character classes

Perl supports the POSIX notation for character classes, which uses names enclosed by [: and :] within the enclosing square brackets. ***STklos*** , thanks to PCRE, also supports this notation. For example,

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

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

alnum	letters and digits
alpha	letters
ascii	character codes 0 - 127
blank	space or tab only
cntrl	control characters
digit	decimal digits (same as \d
graph	printing characters, excluding space
lower	lower case letters
print	printing characters, including space
punct	printing characters, excluding letters and digits
space	white space (not quite the same as \s)
upper	upper case letters

word"	*word* characters (same as \w)
xdigit	hexadecimal digits

The *space* characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to $\backslash$s, which does not include VT (for Perl compatibility).

The name *word* is a Perl extension, and *blank* is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a `^` character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. ***STklos*** (and Perl) also recognize the POSIX syntax `[.ch.]` and `[=ch=]` where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

5.6. Vertical Bar

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

5.7. Internal Option Setting

The settings of the *caseless*, *multiline*, *dotall*, and *EXTENDED* options can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

i	for *caseless*
m	for *multiline*
s	for *dotall*
x	for *extended*

For example, `(?im)` sets *caseless, multiline matching*. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as `(?im-sx)`, which sets *caseless* and *multiline* while unsetting *dotall* and *extended*, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options

An option change within a subpattern affects only that part of the current pattern that follows it, so

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

matches `abc` and `aBc` and no other strings (assuming *caseless* is not used).By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

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

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options *ungreedy* and *extra* can be changed in the same way as the Perl-compatible options by using the characters `U` and `X` respectively. The `(?X)` flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

5.8. Subpatterns

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

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

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

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

- It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, that portion of the subject string that matched the subpattern is set so that it can be used in the regexp-replace or regexp-replace-all functions. 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, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

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

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535, and the maximum depth of nesting of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":".
Thus the two patterns

```
(?i:saturday|sunday)
```

and

```
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

5.9. Named Subpatterns

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with the difficulty, PCRE supports the naming of subpatterns, something that Perl does not provide. The Python syntax (?P<name>...) is used. Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

5.10. Repetition

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

- a literal data character
- the `.` metacharacter
- the `\C` escape sequence
- escapes such as `\d` that match single characters
- a character class

- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)

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?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

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.

If the *ungreedy* option is set (an option which is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with `.*` or `.{0,}` and the *dotall* option (equivalent to Perl's `/s`) is set, thus allowing the `.` to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by `\A`.

In cases where it is known that the subject string contains no newlines, it is worth setting *dotall* in order to obtain this optimization, or alternatively using `^` to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When `.*` is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail, and a later one succeed. Consider, for example:

```
(.*)abc\1
```

If the subject is `"xyz123abc123"` the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
(a|(b))+
```

5.11. Atomic Grouping And Possessive Quantifiers

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher would give up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with (`?>` as in this example:)

```
(?>\d+)foo
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional `+` character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++bar
```

Possessive quantifiers are always greedy; the setting of the *ungreedy* option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun’s Java package.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried. (The example used `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed to

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

5.12. Back References

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see below for a way of doing that). So the pattern

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

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful 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.

Back references to named subpatterns use the Python syntax `(?P=name)`. We could rewrite the above example as follows:

```
(?<p1>(?i)rah)\s+(?P=p1)
```

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the *extended* option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

5.13. 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 `\b`, `\B`, `\A`, `\G`, `\Z`, `\z`, `^` 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.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with `(?!)` because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

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 (at least for 5.8), 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.

Atomic groups can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

or, equivalently,

```
^.*+(?<=abcd)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

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 fInotfR 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 can be nested in any combination. For example,

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

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

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

is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

5.14. Conditional Subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are three kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains non-significant white space to make it more readable (assume the *extended* option) and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is the string `®`, it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false. This is a PCRE extension. See PCRE documentation for recursive patterns.

If the condition is not a sequence of digits or `®`, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=`(^a-z)*`(a-z))
\d{2}-`(a-z){3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

5.15. Comments

The sequence `(?#` marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the *extended* option is set, an unescaped `#` character outside a character class introduces a comment that continues up to the next newline character in the pattern.

5.16. Subpatterns As Subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern

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

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

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

is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

5.17. Regexp Procedures

This section lists the ***STklos*** functions that can use PCRE regexpr described before

STklos procedure

```
(string→regexp string)
```

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

STklos procedure

```
(regexp? obj)
```

Regexp returns #t if obj is a regexp value created by the regexp, otherwise regexp returns #f.

STklos procedure

```
(regexp-match pattern str)
(regexp-match-positions pattern str)
```

These functions attempt to match pattern (a string or a regexp value) to str. If the match fails, #f is returned. If the match succeeds, a list (containing strings for regexp-match and positions for regexp-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-positions "(?<=\\d{3})(?<!999)foo"
                        "999foo and 123foo")
                 => ((14 17))
```

STklos procedure

```
(regexp-replace pattern string substitution)
(regexp-replace-all pattern string substitution)
```

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"
```

STklos procedure

```
(regexp-quote str)
```

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.

Chapter 6. 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*** "pattern matching compiler" provides this [1]. The code (and documentation) included in ***STklos*** has been stolen from the Bigloo package v4.5 (the only difference between both package is the pattern matching of structures which is absent in ***STklos***.

The technique used is described in details in C. Queinnec and J-M. Geffroy paper [QuG92], and the code generated can be considered optimal

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 `a`s followed by zero or more ``` ` ````b`s.
- Pattern matching on lists as well as on vectors.

6.1. The Pattern Language

The syntax for `<pattern>` is:

[:small]

<pattern>	**Matches:**
<atom>	the <atom>.
(kwote <atom>)	any expression eq? to <atom>.
(and <pat$_1$> … <pat$_n$>)	if all of <pati> match.
(or <pat$_1$> … …<pat$_n$>)	if any of <pat$_1$> through <pat$_n$> matches.
(not <pat>)	if <pat> doesn’t match.
(? <predicate>)	if <predicate> is true.
(<pat$_1$> … <pat$_n$>)	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> … <pat$_n$>)	a vector of n elements.
?<id>	anything, and binds id as a variable.
?-	anything.

<pattern>	Matches:
??-	any (possibly empty) repetition of anything in a list.
???-	any end of list.

Remark: `and`, `or`, `not` 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.

`??-` 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`.

6.2. *STklos* Pattern Matching Facilities

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

STklos syntax

```
(match-case <key> <clause> …)
```

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

```
(<pattern> <expression> ...)
```

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 `<expression>`s in that clause are evaluated from left to right in an environment where the pattern variables are bound to the corresponding subparts of `<key>`, 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 `<key>`, 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 for tests is `eq?`.

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

(match-case '(a (b c) d)
   ((?x ?y) (list 'length=2 y x))
   ((?x ?y ?z) (list 'length=3 z y x)))
                        => (length=3 d (b c) a)
```

STklos syntax

```
(match-lambda <clause> ...)
```

`match-lambda` 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
```

[1] The "pattern matching compiler" has been written by Jean-Marie Geffroy and is part of the Manuel Serrano's Bigloo compiler since several years [Bigloo]

Chapter 7. Exceptions and Conditions

7.1. Exceptions

The following text is extracted from **SRFI-34** (*Exception Handling for Programs*), from which ***STklos*** exceptions are derived. Note that exceptions are now part of R^{7}RS.

Exception handlers are one-argument procedures that determine the action the program takes when an exceptional situation is signalled. The system implicitly maintains a current exception handler.

The program raises an exception by invoking the current exception handler, passing to it an object encapsulating information about the exception. Any procedure accepting one argument may serve as an exception handler and any object may be used to represent an exception.

The system maintains the current exception handler as part of the dynamic environment of the program, akin to the current input or output port, or the context for dynamic-wind. The dynamic environment can be thought of as that part of a continuation that does not specify the destination of any returned values. It includes the current input and output ports, the dynamic-wind context, and this SRFI's current exception handler.

STklos syntax

```
(with-handler <handler> <expr₁> … <exprₙ>)
```

Evaluates the sequences of expressions `<expr₁>` to `<exprₙ>`. `<handler>` must be a procedure that accepts one argument. It is installed as the current exception handler for the dynamic extent (as determined by dynamic-wind) of the evaluations of the expressions

```
(with-handler (lambda (c)
                (display "Catch an error\\n"))
   (display "One ... ")
   (+ "will yield" "an error")
   (display "... Two"))
       |- "One ... Catch an error"
```

R^{7}RS procedure

```
(with-exception-handler <handler> <thunk>)
```

This form is similar to `with-handler`. It uses a ***thunk*** instead of a sequence of expressions. It is conform to **SRFI-34** (*Exception Handling for Programs*). In fact,

```
(with-handler <handler> <expr1> ... <exprn>)
```

is equivalent to

```
(with-exception-handler <handler>
  (lambda () <expr1> ... <exprn>))
```

R⁷RS procedure

```
(raise obj)
```

Invokes the current exception handler on `obj`. The handler is called in the dynamic environment of the call to `raise`, except that the current exception handler is that in place for the call to `with-handler` that installed the handler being called.

```
(with-handler (lambda (c)
             (format "value ~A was raised" c))
   (raise 'foo)
   (format #t "never printed\\n"))
          => "value foo was raised"
```

R⁷RS procedure

```
(raise-continuable obj)
```

Raises an exception by invoking the current exception handler on `obj`. The handler is called with the same dynamic environment as the call to `raise-continuable`, except that: (1) the current exception handler is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to `raise-continuable`.

```
(with-exception-handler
  (lambda (con)
    (cond
      ((string? con)
       (display con))
      (else
       (display "a warning has been issued")))
    42)
  (lambda ()
    (+ (raise-continuable "should be a number")
       23)))
  ;; prints should be a number
                => 65
```

R7RS procedure

```
(guard (<var> <clause1 > <clause2 > …) <body>)
```

Evaluating a guard form evaluates `<body>` with an exception handler that binds the raised object to `<var>` and within the scope of that binding evaluates the clauses as if they were the clauses of a cond expression. That implicit cond expression is evaluated with the continuation and dynamic environment of the `guard` expression. If every `<clause>`s test evaluates to false and there is no `else` clause, then `raise` is re-invoked on the raised object within the dynamic environment of the original call to `raise` except that the current exception handler is that of the `guard` expression.

```
(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'a 42))))
         => 42

(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'b 23))))
         => (b . 23)

(with-handler (lambda (c) (format "value ~A was raised" c))
  (guard (condition
       ((assq 'a condition) => cdr)
       ((assq 'b condition)))
      (raise (list (cons 'x 0)))))
         => "value ((x . 0)) was raised"
```

STklos procedure

```
(current-exception-handler)
```

Returns the current exception handler. This procedure is defined in ,(link-srfi 18).

7.2. Conditions

The following text is extracted from **SRFI-35** (*Conditions*), from which ***STklos*** conditions are derived.

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

Conditions available in ***STklos*** are derived from **SRFI-35** and in this SRFI two mechanisms to enable this kind of communication are provided:

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

Conditions are structures with named slots. Each condition belongs to one condition type (a condition type can be made from several condition types). Each condition type specifies a set of slot names. A condition belonging to a condition type includes a value for each of the type's slot names. These values can be extracted from the condition by using the appropriate slot name.

There is a tree of condition types with the distinguished `&condition` as its root. All other condition types have a parent condition type.

Conditions are implemented with ***STklos*** structures (with a special bit indicating that there are conditions). Of course, condition types are implemented with structure types. As a consequence, functions on structures or structures types are available on conditions or conditions types (the contrary is not true). For instance, if C is a condition, the expression

```
(struct->list C)
```

is a simple way to see it's slots and their associated value.

STklos procedure

```
(make-condition-type id parent slot-names)
```

`Make-condition-type` returns a new condition type. `Id` must be a symbol that serves as a symbolic name for the condition type. `Parent` must itself be a condition type. `Slot-names` must be a list of symbols. It identifies the slots of the conditions associated with the condition type.

STklos procedure

```
(condition-type? obj)
```

Returns `#t` if `obj` is a condition type, and `#f` otherwise

STklos procedure

```
(make-compound-condition-type id ct₁ …)
```

`Make-compound-condition-type` returns a new condition type, built from the condition types `ct₁`, … `Id` must be a symbol that serves as a symbolic name for the condition type. The slots names of the new condition type is the union of the slots of conditions `ct₁` …

This function is not defined in **SRFI-34** (*Exception Handling for Programs*).

STklos procedure

```
(make-condition type slot-name value …)
```

`Make-condition` creates a condition value belonging condition type `type`. The following arguments must be, in turn, a slot name and an arbitrary value. There must be such a pair for each slot of `type` and its direct and indirect supertypes. `Make-condition` returns the condition value, with the argument values associated with their respective slots.

```
(let* ((ct (make-condition-type 'ct1 &condition '(a b)))
       (c  (make-condition ct 'b 2 'a 1)))
  (struct->list c))
     => ((a . 1) (b . 2))
```

STklos procedure

```
(condition? obj)
```

Returns `#t` if `obj` is a condition, and `#f` otherwise

STklos procedure

```
(condition-has-type? condition condition-type)
```

`Condition-has-type?` tests if `condition` belongs to `condition-type`. It returns `#t` if any of condition 's types includes `condition-type` either directly or as an ancestor and `#f` otherwise.

```
(let* ((ct1 (make-condition-type 'ct1 &condition '(a b)))
       (ct2 (make-condition-type 'ct2 ct1 '(c)))
       (ct3 (make-condition-type 'ct3 &condition '(x y z)))
       (c   (make-condition ct2 'a 1 'b 2 'c 3)))
 (list (condition-has-type? c ct1)
    (condition-has-type? c ct2)
    (condition-has-type? c ct3)))
   => (#t #t #f)
```

STklos procedure

```
(condition-ref condition slot-name)
```

`Condition` must be a condition, and `slot-name` a symbol. Moreover, `condition` must belong to a condition type which has a slot name called `slot-name`, or one of its (direct or indirect) supertypes must have the slot. `Condition-ref` returns the value associated with `slot-name`.

```
(let* ((ct (make-condition-type 'ct1 &condition '(a b)))
       (c  (make-condition ct 'b 2 'a 1)))
  (condition-ref c 'b))
     => 2
```

STklos procedure

```
(condition-set! condition slot-name obj)
```

`Condition` must be a condition, and `slot-name` a symbol. Moreover, `condition` must belong to a condition type which has a slot name called `slot-name`, or one of its (direct or indirect) supertypes must have the slot. `Condition-set!` change the value associated with `slot-name` to `obj`.

Whereas `condition-ref` is defined in ,(srfi 35), `confition-set!` is not.

STklos procedure

```
(make-compound-condition condition0 condition1 …)
```

Make-compound-condition returns a compound condition belonging to all condition types that the condition$_i$ belong to.

Condition-ref, when applied to a compound condition will return the value from the first of the conditioni that has such a slot.

STklos procedure

```
(extract-condition condition condition-type)
```

Condition must be a condition belonging to condition-type. Extract-condition returns a condition of condition-type with the slot values specified by condition. The new condition is always allocated.

```
(let* ((ct1 (make-condition-type 'ct1 &condition '(a b)))
       (ct2 (make-condition-type 'ct2 ct1 '(c)))
       (c2  (make-condition ct2 'a 1 ' b 2 'c 3))
       (c1  (extract-condition c2 ct1)))
  (list (condition-has-type? c1 ct2)
     (condition-has-type? c1 ct1)))
      => (#f #t)
```

7.3. Predefined Conditions

STklos implements all the conditions types which are defined in **SRFI-35** (*Conditions*) and **SRFI-36** (*I/O Conditions*). However, the access functions which are (implicitely) defined in those SRFIs are only available if the file "conditions" is required. This can be done with the call:

```
(require "conditions")
```

Another way to have access to the hierarchy of the **SRFI-35** (*Conditions*) and **SRFI-36** (*I/O Conditions*) condition:

```
(require-extension conditions)
```

The following hierarchy of conditions is predefined:

```
&condition
   &message (has "message" slot)
   &serious
   &error
```

```
     &error-message (has *message*, "location" and "backtrace" slots)
     &i/o-error
        &i/o-port-error (has a "port" slot)
           &i/o-read-error
           &i/o-write-error
           &i/o-closed-error
       &i/o-filename-error (has a "filename" slots)
           &i/o-malformed-filename-error
           &i/o-file-protection-error
               &i/o-file-is-read-only-error
           &i/o-file-already-exists-error
           &i/o-no-such-file-error
    &read-error (has the "line", "column", "position" and "span" slots)
```

Chapter 8. STklos Object System

8.1. Introduction

The aim of this chapter is to present ***STklos*** object system. Briefly stated, ***STklos*** gives the programmer an extensive object system with meta-classes, multiple inheritance, generic functions and multi-methods. Furthermore, its implementation relies on a MOP (Meta Object Protocol (MOP) [AMOP]), in the spirit of the one defined for CLOS [CLtL2].

STklos implementation is derived from the version 1.3 of ***Tiny CLOS***, a pure and clean CLOS-like MOP implementation in Scheme written by Gregor Kickzales [Tiny-Clos]. However, Tiny CLOS implementation was designed as a pedagogical tool and consequently, completeness and efficiency were not the author concern for it. ***STklos*** extends the Tiny CLOS model to be efficient and as close as possible to CLOS, the Common Lisp Object System [CLtL2]. Some features of ***STklos*** are also issued from [Dylan] or [SOS].

This chapter is divided in three parts, which have a quite different audience in mind:

- The first part presents the ***STklos*** object system rather informally; it is intended to be a tutorial of the language and is for people who want to have an idea of the ***look and feel*** of ***STklos***.
- The second part describes the ***STklos*** object system at the ***external*** level (i.e. without requiring the use of the Meta Object Protocol).
- The third and last part describes the ***STklos*** Meta Object Protocol. It is intended for people whio want to play with meta programming.

8.2. Object System Tutorial

The ***STklos*** object system relies on classes like most of the current OO languages. Furthermore, ***STklos*** provides meta-classes, multiple inheritance, generic functions and multi-methods as in CLOS, the Common Lisp Object System [CLtL2] or [Dylan]. This chapter presents ***STklos*** in a rather informal manner. Its intent is to give the reader an idea of the "***look and feel***" of ***STklos*** programming. However, we suppose here that the reader has some basic notions of OO programming, and is familiar with terms such as ***classes, instances*** or ***methods.***

8.2.1. Class definition and instantiation

Class definition

A new class is defined with the `define-class` form. The syntax of `define-class` is close to CLOS `defclass`:

```
(define-class class (superclass~1~ superclass~2~ ...)
  (slot-description1
   slot-description2
   ...)
  metaclass option)
```

The ***metaclass option*** will not be discussed here. The ***superclass***es list specifies the super classes of ***class*** (see Section 8.2.2 for details).

A ***slot description*** gives the name of a slot and, eventually, some *properties* of this slot (such as its initial value, the function which permit to access its value, ...). Slot descriptions will be discussed in ,(index "slot-definition").

As an example, consider now that we want to define a point as an object. This can be done with the following class definition:

```
(define-class <point> ()
  (x y))
```

This definition binds the symbol `<point>` to a new class whose instances contain two slots. These slots are called `x` an `y` and we suppose here that they contain the coordinates of a 2D point.

Let us define now a circle, as a 2D point and a radius:

```
(define-class <circle> (<point>)
  (radius))
```

As we can see here, the class `<circle>` is constructed by inheriting from the class `<point>` and adding a new slot (the `radius` slot).

Instance creation and slot access

Creation of an instance of a previously defined class can be done with the `make` procedure. This procedure takes one mandatory parameter which is the class of the instance which must be created and a list of optional arguments. Optional arguments are generally used to initialize some slots of the newly created instance. For instance, the following form:

```
(define  c (make <circle>))
```

creates a new `<circle>` object and binds it to the `c` Scheme variable.

Accessing the slots of the newly created circle can be done with the `slot-ref` and the `slot-set!` primitives. The `slot-set!` primitive permits to set the value of an object slot and `slot-ref` permits to get its value.

```
(slot-set! c 'x 10)
(slot-set! c 'y 3)
(slot-ref c 'x)       =>  10
(slot-ref c 'y)       =>  3
```

Using the `describe` function is a simple way to see all the slots of an object at one time: this function prints all the slots of an object on the standard output. For instance, the expression:

```
(describe c)
```

prints the following information on the standard output:

```
#[<circle> 81aa1f8] is an an instance of class <circle>.
Slots are:
     radius = #[unbound]
     x = 10
     y = 3
```

Slot Definition

When specifying a slot, a set of options can be given to the system. Each option is specified with a keyword. For instance,

- **:init-form** can be used to supply a default value for the slot.
- **:init-keyword** can be used to specify the keyword used for initializing a slot.
- **:getter** can be used to define the name of the slot getter
- **:setter** can be used to define the name of the slot setter
- **:accessor** can be used to define the name of the slot accessor (see below)

To illustrate slot description, we redefine here the `<point>` class seen before. A new definition of this class could be:

```
(define-class <point> ()
  ((x :init-form 0 :getter get-x :setter set-x! :init-keyword :x)
   (y :init-form 0 :getter get-y :setter set-y! :init-keyword :y)))
```

With this definition,

1. the `x` and `y` slots are set to 0 by default.
2. The value of a slot can also be specified by calling `make` with the `:x` and `:y` keywords.
3. Furthermore, the generic functions `get-x` and `set-x!` (resp. `get-y` and `set-y!`) are automatically defined by the system to read and write the `x` (resp. `y`) slot.

```
(define p1 (make <point> :x 1 :y 2))
(get-x p1)        => 1
(set-x! p1  12)
(get-x p1)        => 12

(define  p2 (make <point> :x 2))
(get-x p2)         => 2
(get-y p2)         => 0
```

Accessors provide an uniform access for reading and writing an object slot. Writing a slot is done with an extended form of `set!` which is close to the Common Lisp `setf` macro. A slot accessor can be defined with the `:accessor` option in the slot description. Hereafter, is another definition of our `<point>` class, using an accessor:

```
(define-class <point> ()
  ((x :init-form 0 :accessor x-of :init-keyword :x)
   (y :init-form 0 :accessor y-of :init-keyword :y)))
```

Using this class definition, reading the x coordinate of the `p` point can be done with:

```
(x-of p)
```

and setting it to 100 can be done using the extended `set!`

```
(set! (x-of p) 100)
```

STklos also define `slot-set!` as the setter function of `slot-ref`. As a consequence, we have

```
(set! (slot-ref p 'y) 100)
(slot-ref p 'y)       => 100
```

Virtual Slots

Suppose that we need slot named `area` in circle objects which contain the area of the circle. One way to do this would be to add the new slot to the class definition and have an initialisation form for this slot which takes into account the radius of the circle. The problem with this approach is that if the `radius` slot is changed, we need to change `area` (and vice-versa). This is something which is hard to manage and if we don’t care, it is easy to have a `area` and `radius` in an instance which are "un-synchronized". The virtual slot mechanism avoid this problem.

A virtual slot is a special slot whose value is calculated rather than stored in an object. The way to read and write such a slot must be given when the slot is defined with the `:slot-ref` and `:slot-set!` slot options.

A complete definition of the `<circle>` class using virtual slots could be:

```
(define-class <circle> (<point>)
  ((radius :init-form 0 :accessor radius :init-keyword :radius)
   (area :allocation :virtual :accessor area
         :slot-ref (lambda (o) (let ((r (radius o))) (* 3.14 r r)))
         :slot-set! (lambda (o v) (set! (radius o) (sqrt (/ v 3.14)))))))
```

Here is an example using this definition of `<circle>`

```
(define c (make <circle> :radius 1))
(radius c)                          => 1
(area c)                            => 3.14
(set! (area x) (* 4 (area x)))
(area c)                            => 12.56   ;; (i.e. □π)
(radius c)                          => 2.0
```

Of course, we can also used the function `describe` to visualize the slots of a given object. Applied to the prvious `c`, it prints:

```
#[<circle> 81b2348] is an an instance of class <circle>.
Slots are:
     area = 12.56
     radius = 2.0
     x = 0
     y = 0
```

8.2.2. Inheritance

Class hierarchy and inheritance of slots

inheritance Inheritance is specified upon class definition. As said in the introduction, ***STklos*** supports multiple inheritance. Hereafter are some classes definition:

```
(define-class A () (a))
(define-class B () (b))
(define-class C () (c))
(define-class D (A B) (d a))
(define-class E (A C) (e c))
(define-class F (D E) (f))
```

Here,

- `A`, `B`, `C` have a null list of super classes. In this case, the system will replace it by the list which only contains `<object>`, the root of all the classes defined by `define-class`.
- `D`, `E`, and `F` use multiple inheritance: each class inherits from two previously defined classes. Those class definitions define a hierarchy which is shown in Figure 1.

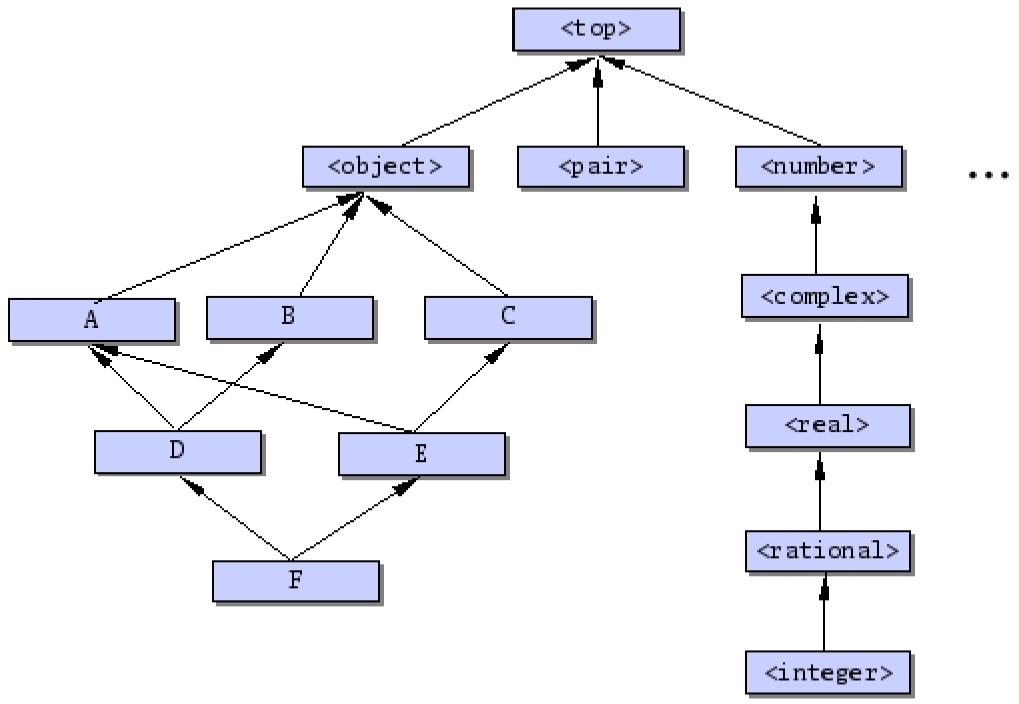

Figure 1. A class hiearchy

In this figure, the class `<top>` is also shown; this class is the super class of all Scheme objects. In particular, `<top>` is the super class of all standard Scheme types.

The set of slots of a given class is calculated by "unioning" the slots of all its super class. For instance, each instance of the class `D` defined before will have three slots (`a`, `b` and `d`). The slots of a class can be obtained by the `class-slots` primitive. For instance,

```
(class-slots A) => (a)
(class-slots E) => (a e c)
(class-slots F) => (b e c d a f)
```

The order of slots is not significant.

Class precedence list

A class may have more than one superclass [1].

With single inheritance (only one superclass), it is easy to order the super classes from most to least specific. This is the rule:

Rule 1: Each class is more specific than its superclasses.

With multiple inheritance, ordering is harder. Suppose we have

```
(define-class X ()
   ((x :init-form 1)))
```

```
(define-class Y ()
   ((x :init-form 2)))

(define-class Z (X Y)
   (z :init-form 3))
```

In this case, given **Rule 1**, the `Z` class is more specific than the `X` or `Y` class for instances of `Z`. However, the `:init-form` specified in `X` and `Y` leads to a problem: which one overrides the other? Or, stated differently, which is the default initial value of the `x` slot of a `Z` instance. The rule in ***STklos***, as in CLOS, is that the superclasses listed earlier are more specific than those listed later. So:

> **Rule 2: For a given class, superclasses listed earlier are more specific than those listed later.**

These rules are used to compute a linear order for a class and all its superclasses, from most specific to least specific. This order is called the "***class precedence list***" of the class. Given these two rules, we can claim that the initial form for the `x` slot of previous example is 1 since the class `X` is placed before `Y` in the super classes of `Z`. These two rules are not always sufficient to determine a unique order. However, they give an idea of how the things work. ***STklos*** algorithm for calculating the class precedence list of a class is a little simpler than the CLOS one described in [AMOP] for breaking ties. Consequently, the calculated class precedence list by ***STklos*** algorithm can be different than the one given by the CLOS one in some subtle situations. Taking the `F` class shown in Figure 1, the ***STklos*** calculated class precedence list is

```
(F D E A B C <object> <top>)
```

whereas it would be the following list with a CLOS-like algorithm:

```
(F D E A C B <object> <top>)
```

However, it is usually considered a bad idea for programmers to rely on exactly what the order is. If the order for some superclasses is important, it can be expressed directly in the class definition. The precedence list of a class can be obtained by the function `class-precedence-list`. This function returns a ordered list whose first element is the most specific class. For instance,

```
(class-precedence-list D)
    => (#[<class> D 81aebb8] #[<class> A 81aab88]
        #[<class> B 81aa720] #[<class> <object> 80eff90]
        #[<class> <top> 80effa8])
```

However, this result is hard to read; using the function `class-name` yields a clearer result:

```
(map class-name (class-precedence-list D))
```

```
   => (D A B <object> <top>)
```

8.2.3. Generic function

Generic functions and methods

Neither ***STklos*** nor CLOS use the message passing mechanism for methods as most Object Oriented languages do. Instead, they use the notion of ***generic function***.A generic function can be seen as a "***tanker***" of methods. When the evaluator requests the application of a generic function, all the applicable methods of this generic function will be grabbed and the most specific among them will be applied. We say that a method `M` is ***more specific*** than a method `M'` if the class of its parameters are more specific than the `M'` ones. To be more precise, when a generic function must be "***called***" the system

- searchs among all the generic function methods those which are applicable (i.e. the ones which filter on types which are ***compatible*** with the actual argument list),
- sorts the list of applicable methods in the "***most specific***" order,
- calls the most specific method of this list (i.e. the first of the list of sorted methods).

The definition of a generic function is done with the `define-generic` macro. Definition of a new method is done with the `define-method` macro.

Consider the following definitions:

```
(define-generic M)
(define-method M((a <integer>) b) 'integer)
(define-method M((a <real>)    b) 'real)
(define-method M(a b)             'top)
```

The `define-generic` call defines `M` as a generic function. Note that the signature of the generic function is not given upon definition, contrarily to CLOS. This permits methods with different signatures for a given generic function, as we shall see later. The three next lines define methods for the `M` generic function. Each method uses a sequence of ***parameter specializers*** that specify when the given method is applicable. A specializer permits to indicate the class a parameter must belong (directly or indirectly) to be applicable. If no specializer is given, the system defaults it to `<top>`>. Thus, the first method definition is equivalent to

```
(define-method M((a <integer>) (b <top>)) 'integer)
```

Now, let us look at some possible calls to generic function `M`:

```
(M 2 3)      => integer
(M 2 #t)     => integer
(M 1.2 'a)   => real
(M #t #f)    => top
```

```
(M 1 2 3)    => error (no method with 3 parameters)
```

The preceding methods use only one specializer per parameter list. Of course, each parameter can use a specializer. In this case, the parameter list is scanned from left to right to determine the applicability of a method. Suppose we declare now

```
`(define-method M ((a <integer>) (b <number>))
    'integer-number)

(define-method M ((a <integer>) (b <real>))
    'integer-real)

(define-method M (a (b <number>))
    'top-number)

(define-method M (a b c)
    'three-parameters)
```

In this case, we have

```
(M 1 2)     => integer-integer
(M 1 1.0)   => integer-real
(M 'a 1)    => top-number
(M 1 2 3)   => three-parameters
```

- Before defining a new generic function `define-generic`, verifies if the symbol given as parameter is already bound to a procedure in the current environment. If so, this procedure is added, as a method to the newly created generic function. For instance:

  ```
  (define-generic log)  ; transform "log" in a generic function

  (define-method log ((s <string>) . l)
     (apply format  (current-error-port) s l)
     (newline (current-error-port)))

  (log "Hello, ~a" "world")      |- Hello, world
  (log 1)                        => 0 ; standard "log" procedure
  ```

- `define-method` automatically defines the generic function if it has not been defined before. Consequently, most of the time, the `define-generic` is not needed.

Next-method

When a generic function is called, the list of applicable methods is built. As mentioned before, the most specific method of this list is applied (see Section 8.2.3).

This method may call, if needed, the next method in the list of applicable methods. This is done by using the special form next-method. Consider the following definitions

```
(define-method Test((a <integer>))
   (cons 'integer (next-method)))

(define-method Test((a <number>))
   (cons 'number  (next-method)))

(define-method Test(a)
   (list 'top))
```

With those definitions, we have:

```
(Test 1)     => (integer number top)
(Test 1.0)   => (number top)
(Test #t)    => (top)
```

Standard generic functions

Printing objects

When the Scheme primitives write or display are called with a parameter which is an object, the write-object or display-object generic functions are called with this object and the port to which the printing must be done as parameters. This facility permits to define a customized printing for a class of objects by simply defining a new method for this class. So, defining a new printing method overloads the standard printing method (which just prints the class of the object and its hexadecimal address).

For instance, we can define a customized printing for the <point> used before as:

```
(define-method display-object ((p <point>) port)
  (format port "<Point x=~S y=~S>" (slot-ref p 'x) (slot-ref p 'y)))
```

With this definition, we have

```
(define p (make <point> :x 1 :y 2))
(display p)                        |= <Point x=1 y=2>
```

The Scheme primitive write tries to write objects, in such a way that they are readable back with the read primitive. Consequently, we can define the writing of a <point> as a form which, when read, will build back this point:

```
(define-method write-object ((p <point>) port)
 (format port "#,(make <point> :x ~S :y ~S)"
              (get-x p) (get-y p)))
```

With this method, writing the `p` point defined before prints the following text on the output port:

```
#,(make <point> :x 1 :y 2)
```

Note here the usage of the `#,` notation of **SRFI-10** (*Sharp Comma External Form*) used here to "evaluate" the form when reading it. We suppose here that we are in a context where we already defined:

```
(define-reader-ctor 'make (lambda l (eval `(make ,@l))))
```

Comparing objects
When objects are compared with the `eqv?` or `equal?` Scheme standard primitives, ***STklos*** calls the `object-eqv?` or `object-equal?` generic functions. This facility permits to define a customized comparison function for a class of objects by simply defining a new method for this class. Defining a new comparison method overloads the standard comparaison method (which always returns `#f`). For instance we could define the following method to compare points:

```
(define-method object-eqv? ((a <point>) (b <point>))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))
```

8.3. Object System Reference

This section needs to be written.

8.4. Main Functions and sytaxes

STklos syntax

```
(define-class name supers slots . options)
```

Creates a class whose name is `name`, and whose superclasses are in the list `supers`, with the slots specified by the list `slots`.

As an example, this is the definition of a point:

```
(define-class <point> ()
  (x y))
```

In another example, a class `<circle>` that inherits `<point>`.

```
(define-class <circle> (<point>)
  (radius))
```

The following options can be passed to slots:

- `:init-form` is the default value for the slot.
- `:init-keyword` is the keyword for initializing the slot.
- `:getter` is the name of the getter method.
- `:setter` is the name of the setter method.
- `:accessor` is the name of the accessor (setter and getter) method.

For example,

```
(define-class <point> ()
  (x :init-form 0 :getter get-x :setter set-x! :init-keyword :x)
  (y :init-form 0 :getter get-y :setter set-y! :init-keyword :y))
```

STklos also defines setters for the specified getters, so the following will work with the definition of `<point>` given above:

```
(set! (slot-ref my-point 'x) 50)
```

Accessors, are methods which can be used as getter and setter, as shown bellow

```
(define-class <circle> (<point>)
  ((radius :accessor radius :init-keyword :radius)))

(define x (make <circle> :radius 100))
(radius x)                              => 100
(set! (radius x) 200)
(radius x)                              => 200
```

STklos procedure

```
(find-class name)
```

```
(find-class name default)
```

Returns the class whose name is equal to symbol `name`. If `name` is not a class instance, the `default` value is returned, if present.

STklos procedure

```
(is-a? obj class)
```

Returns `#t` if `obj` is an instance of `class`, and `#f` otherwise.

[1] This section is an adaptation of Jeff Dalton's (J.Dalton@ed.ac.uk) "***Brief introduction to CLOS***" which can be found at http://www.aiai.ed.ac.uk/~jeff/clos-guide.html

Chapter 9. Threads, Mutexes and Condition Variables

The thread system provides the following data types:

- Thread (a virtual processor which shares object space with all other threads)
- Mutex (a mutual exclusion device, also known as a lock and binary semaphore)
- Condition variable (a set of blocked threads)

The ***STklos*** thread system is conform to **SRFI-18** (*Multithreading support*), and implement all the SRFI mechanisms. See this SRFI documentation for a more complete description

9.1. Threads

STklos procedure

```
(make-thread thunk)
(make-thread thunk name)
(make-thread thunk name stack-size)
```

Returns a new thread. This thread is not automatically made runnable (the procedure `thread-start!` must be used for this). A thread has the following fields: name, specific, end-result, end-exception, and a list of locked/owned mutexes it owns. The thread's execution consists of a call to thunk with the "initial continuation". This continuation causes the (then) current thread to store the result in its end-result field, abandon all mutexes it owns, and finally terminate. The dynamic-wind stack of the initial continuation is empty. The optional name is an arbitrary Scheme object which identifies the thread (useful for debugging); it defaults to an unspecified value. The specific field is set to an unspecified value. The thread inherits the dynamic environment from the current thread. Moreover, in this dynamic environment the exception handler is bound to the "initial exception handler" which is a unary procedure which causes the (then) current thread to store in its end-exception field an "uncaught exception" object whose "reason" is the argument of the handler, abandon all mutexes it owns, and finally terminate.

The optional parameter `stack-size` permits to specify the size (in words) reserved for the thread. This option does not exist in **SRFI-18**.

STklos procedure

```
(current-thread)
```

Returns the current thread.

```
(eq? (current-thread) (current-thread))  =>  #t
```

STklos procedure

```
(thread-start! thread)
```

Makes `thread` runnable. The `thread` must be a new thread. `Thread-start!` returns the thread.

```
(let ((t (thread-start! (make-thread
                          (lambda () (write 'a))))))
   (write 'b)
   (thread-join! t))       =>  unspecified
                               after writing ab or ba
```

STklos procedure

```
(thread-yield!)
```

The current thread exits the running state as if its quantum had expired. `Thread-yield!` returns an unspecified value.

STklos procedure

```
(thread-terminate! thread)
```

Causes an abnormal termination of the `thread`. If the `thread` is not already terminated, all mutexes owned by the `thread` become unlocked/abandoned and a "terminated thread exception" object is stored in the thread's end-exception field. If `thread` is the current thread, `thread-terminate!` does not return. Otherwise, `thread-terminate!` returns an unspecified value; the termination of the thread will occur before `thread-terminate!` returns.

- This operation must be used carefully because it terminates a thread abruptly and it is impossible for that thread to perform any kind of cleanup. This may be a problem if the thread is in the middle of a critical section where some structure has been put in an inconsistent state. However, another thread attempting to enter this critical section will raise an "abandoned mutex exception" because the mutex is unlocked/abandoned.
- On *Android*, `thread-terminate!` can be used only to terminate the current

thread. Trying to kill another thread produces an error.

STklos procedure

```
(thread-sleep! timeout)
```

The current thread waits until the `timeout` is reached. This blocks the thread only if timeout represents a point in the future. It is an error for timeout to be `#f`. `Thread-sleep!` returns an unspecified value.

STklos procedure

```
(thread-join! thread)
(thread-join! thread timeout)
(thread-join! thread timeout timeout-val)
```

The current thread waits until the `thread` terminates (normally or not) or until the timeout is reached if `timeout` is supplied. If the timeout is reached, `thread-join!` returns `timeout-val` if it is supplied, otherwise a "join timeout exception" is raised. If the `thread` terminated normally, the content of the end-result field is returned, otherwise the content of the end-exception field is raised.

```
(let ((t (thread-start! (make-thread (lambda ()
                                        (expt 2 100))))))
  (thread-sleep! 1)
  (thread-join! t)) => 1267650600228229401496703205376
```

STklos procedure

```
(thread? obj)
```

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

```
(thread? (current-thread))  => #t
(thread? 'foo)              => #f
```

STklos procedure

```
(thread-name thread)
```

Returns the name of the `thread`.

```
(thread-name (make-thread (lambda () #f) 'foo))  =>  foo
```

STklos procedure

```
(thread-stack-size thread)
```

Returns the allocated stack size for `thread`.

```
(thread-stack-size (make-thread (lambda () #f) 'foo 2000)) => 2000
```

Note that this procedure is not present in **SRFI-18**.

STklos procedure

```
(thread-specific thread)
```

Returns the content of the `thread's specific field.

STklos procedure

```
(thread-specific-set! thread)
```

Stores `obj` into the `thread�s specific field. `Thread-specific-set!` returns an unspecified value.

```
(thread-specific-set! (current-thread) "hello")
           =>  unspecified
(thread-specific (current-thread))
           =>  "hello"
```

9.2. Mutexes

STklos procedure

```
(make-mutex)
(make-mutex name)
```

Returns a new mutex in the unlocked/not-abandoned state. The optional `name` is an arbitrary Scheme object which identifies the mutex (useful for debugging); it defaults to an unspecified value. The mutex's specific field is set to an unspecified value.

STklos procedure

```
(mutex? obj)
```

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

STklos procedure

```
(mutex-name mutex)
```

Returns the name of the `mutex`.

```
(mutex-name (make-mutex 'foo))  =>  foo
```

STklos procedure

```
(mutex-specific mutex)
```

Returns the content of the `mutex's specific field.

STklos procedure

```
(mutex-specific! mutex obj)
```

Stores `obj` into the `mutex's specific field and eturns an unspecified value.

```
(define m (make-mutex))
(mutex-specific-set! m "hello")  =>  unspecified
```

```
(mutex-specific m)               =>  "hello"

(define (mutex-lock-recursively! mutex)
  (if (eq? (mutex-state mutex) (current-thread))
      (let ((n (mutex-specific mutex)))
        (mutex-specific-set! mutex (+ n 1)))
      (begin
        (mutex-lock! mutex)
        (mutex-specific-set! mutex 0))))

(define (mutex-unlock-recursively! mutex)
  (let ((n (mutex-specific mutex)))
    (if (= n 0)
        (mutex-unlock! mutex)
        (mutex-specific-set! mutex (- n 1)))))
```

STklos procedure

```
(mutex-state mutex)
```

Returns information about the state of the `mutex`. The possible results are:

- a thread **T**: the mutex is in the locked/owned state and thread **T** is the owner of the mutex
- the symbol **not-owned**: the mutex is in the locked/not-owned state
- the symbol **abandoned**: the mutex is in the unlocked/abandoned state
- the symbol **not-abandoned**: the mutex is in the unlocked/not-abandoned state

```
(mutex-state (make-mutex))  =>  not-abandoned

(define (thread-alive? thread)
  (let ((mutex (make-mutex)))
    (mutex-lock! mutex #f thread)
    (let ((state (mutex-state mutex)))
      (mutex-unlock! mutex) ; avoid space leak
      (eq? state thread))))
```

STklos procedure

```
(mutex-lock! mutex)
(mutex-lock! mutex timeout)
(mutex-lock! mutex timeout thread)
```

If the mutex is currently locked, the current thread waits until the mutex is unlocked, or until the timeout is reached if timeout is supplied. If the timeout is reached, mutex-lock! returns #f. Otherwise, the state of the mutex is changed as follows:

- if thread is #f the mutex becomes *locked/not-owned*,
- otherwise, let T be thread (or the current thread if thread is not supplied),
  - if T is terminated the mutex becomes *unlocked/abandoned*,
  - otherwise mutex becomes *locked/owned* with T as the owner.

After changing the state of the mutex, an "abandoned mutex exception" is raised if the mutex was unlocked/abandoned before the state change, otherwise mutex-lock! returns #t.

```
(define (sleep! timeout)
  ;; an alternate implementation of thread-sleep!
  (let ((m (make-mutex)))
  (mutex-lock! m #f #f)
  (mutex-lock! m timeout #f)))
```

STklos procedure

```
(mutex-unlock! mutex)
(mutex-unlock! mutex condition-variable)
(mutex-unlock! mutex condition-variable timeout)
```

Unlocks the mutex by making it unlocked/not-abandoned. It is not an error to unlock an unlocked mutex and a mutex that is owned by any thread. If condition-variable is supplied, the current thread is blocked and added to the condition-variable before unlocking mutex; the thread can unblock at any time but no later than when an appropriate call to condition-variable-signal! or condition-variable-broadcast! is performed (see below), and no later than the timeout (if timeout is supplied). If there are threads waiting to lock this mutex, the scheduler selects a thread, the mutex becomes locked/owned or locked/not-owned, and the thread is unblocked. mutex-unlock! returns #f when the timeout is reached, otherwise it returns #t.

9.3. Condition Variables

STklos procedure

```
(make-conditon-variable)
(make-conditon-variable name)
```

Returns a new empty condition variable. The optional name is an arbitrary Scheme object which identifies the condition variable (useful for debugging); it defaults to an unspecified value. The condition variable’s specific field is set to an unspecified value.

STklos procedure

```
(conditon-variable? obj)
```

Returns `#t` if `obj` is a condition variable, otherwise returns `#f`.

STklos procedure

```
(conditon-variable-name conditon-variable)
```

Returns the name of the `condition-variable`.

STklos procedure

```
(conditon-variable-specific conditon-variable)
```

Returns the content of the `condition-variable’s specific field.

STklos procedure

```
(conditon-variable-specific-set! conditon-variable obj)
```

Stores `obj` into the `condition-variable’s specific field.

STklos procedure

```
(condition-variable-signal! condition-variable)
```

If there are threads blocked on the `condition-variable`, the scheduler selects a thread and unblocks it. `Condition-variable-signal!` returns an unspecified value.

STklos procedure

```
(condition-variable-broadcast! condition-variable)
```

Unblocks all the threads blocked on the `condition-variable`. `Condition-variable-broadcast!` returns an unspecified value.

9.4. Conditions

STklos procedure

```
(join-timeout-exception? obj)
```

Returns `#t` if `obj` is a *join timeout exception* object, otherwise returns `#f`.

A *join timeout exception* is raised when thread-join! is called, the timeout is reached and no timeout-val is supplied.

STklos procedure

```
(abandoned-mutex-exception? obj)
```

Returns `#t` if `obj` is an *abandoned mutex exception* object, otherwise returns `#f`.

An *abandoned mutex exception* is raised when the current thread locks a mutex that was owned by a thread which terminated ,(see `mutex-lock!`).

STklos procedure

```
(terminated-thread-exception? obj)
```

Returns `#t` if `obj` is a *terminated thread exception* object, otherwise returns `#f`.

A *terminated thread exception* is raised when thread-join! is called and the target thread has terminated as a result of a call to `thread-terminate!`.

STklos procedure

```
(uncaught-exception? obj)
```

Returns `#t` if `obj` is an *uncaught exception* object, otherwise returns `#f`.

An *uncaught exception* is raised when `thread-join!` is called and the target thread has terminated because it raised an exception that called the initial exception handler of that thread.

STklos procedure

```
(uncaught-exception-reason exc)
```

Returns the object which was passed to the initial exception handler of that thread (`exc` must be an *uncaught exception* object).

Chapter 10. STklos Customization

10.1. Parameter Objects

STklos environement can be customized using Parameter Objects. These parmaters are listed below.

STklos procedure

```
(real-precision)
(real-precision value)
```

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
```

STklos procedure

```
(accept-srfi-169-numbers)
(accept-srfi-169-numbers value)
```

This parameter object permits to change the behavior of the reader with underscores in numbers. Numbers with underscores are defined in ,(link-srfi 169). By default, this variable is true, meaning that underscores are accepted in numbers.

```
(accept-srfi-169-numbers)        => #t
(symbol? '1_000_000)             => #f
(number? '1_000_000)             => #t
(accept-srfi-169-numbers #f)
(symbol? '1_000_000)             => #t
(number? '1_000_000)             => #f
```

STklos procedure

```
(read-case-sensitive)
```

```
(read-case-sensitive value)
```

This parameter object permits to change the default behaviour of the `read` primitive when reading a symbol. If this parameter has a true value a symbol is not converted to a default case when interned. Since R^{7}RS requires that symbol are case insignificant, the default value of this parameter is `#t`.

```
(read-case-sensitive)          => `#t`
(read-from-string "ABC")       => ABC
(read-case-sensitive #f)
(read-from-string "ABC")       => abc
```

- Default behaviour can be changed for a whole execution with the `--case -sensitive` or `case-insensitive` options.
- See also syntax for *special characters* in symbols.

STklos procedure

```
(write-pretty-quotes)
(write-pretty-quotes value)
```

This parameter object permits to change the default behaviour of the `display` or `write` primitives when they write a list which starts with the symbol quote, quasiquote, unquote or unquote-splicing. If this parameter has a false value, the writer uses the list notation instead of a more human-readable value. By default, this parameter value is set to `#t`.

```
(let ((x ''a))
  (display x)
  (display " ")
  (write-pretty-quotes #f)
  (display x))               |- 'a (quote a)
```

STklos procedure

```
(load-path)
(load-path value)
```

`load-path` is a parameter object. It 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 ,(emph "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` shell variable.

Giving a `value` to the parameter `load-path` permits to change the current list of paths.

STklos procedure

```
(load-suffixes)
(load-suffixes value)
```

`load-suffixes` is a parameter object. It 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.

STklos procedure

```
(load-verbose)
(load-verbose value)
```

`load-verbose` is a parameter object. It permits to display the path name of the files which are loaded by `load` or `try-load` on the current error port, when set to a true value. If `load-verbose` is set to `#f`, no message is printed.

STklos procedure

```
(thread-handler-error-show)
(thread-handler-error-show value)
```

When an untrapped error occurs in a thread, it produces an *uncaught exception* which can finally be trapped when the thread is *joined*. Setting the `thread-handler-error-show` parameter permits to see error message as soon as possible, even without joining the thread. This makes debugging easier. By default, this parameter is set to `#t`.

STklos procedure

```
(stklos-debug-level)
```

`stklos-debug-level` is a parameter objet. It permits to know the current debugging level. The default value of this parameter is 0 (meaning "no debug"). Note that the debugging level can also be set by the `--debug` option of the `stklos(1)` command.

STklos procedure

```
(repl-theme)
(repl-theme plist)
(repl-theme name)
```

The ***STklos*** REPL colors can be customized using a property list (or a theme name) using the `repl-theme` parameter. The properties that can be used in this property list are:

- `#:error` for the color of error messages
- `#:prompt` for the color of the prompt
- `#:help-prompt` for the color of the help prompt
- `#:help` for the color of the help messages
- `#:repl-depth` for the color of the depth indicator for recursive REPLs
- `#:info` for the color of information messages.

There are three default themes:

- **classic** which is the (colorful) default theme.
- **monochrome** which doesn't use colors
- **minimal** where only the prompt and errors are colored.

Colors are expressed using the conventions of the `ansi-color` primitive. For instance:

```
(key-set! (repl-theme) :prompt '(bold green)) ;; set the prompt to bold green

(repl-theme 'minimal)                          ;; to use a sober prompt

(repl-theme '(#:error (bold red)               ;; explicitly defined theme
              #:prompt (bold bg-red white)))
```

A good place to change your theme is the `.stklosrc` file.

10.2. Environment variables

The following variables can be used to customize ***STklos***:

- **STKLOS_LOAD_PATH**: This is a colon-separated list of directories in which stklos looks for loading files. It is used by primitives such as `load` or `try-load`. See also the `load-path` parameter.

- **STKLOS-FRAME**: This variable must contains an integer which indicates the number of frames printed on an error. Use the value 0 for an unlimited backtrace.
- **STKLOS-CONFDIR**: This variable can be used to designate the directory used by ***STklos*** to store its configuration or packages files. It not set, the default ***STklos*** configuration directory is by default `~/.stklos`.

Chapter 11. The ScmPkg Package System

ScmPkg is a package distribution system for Scheme. It is currently supported by Bigloo and ***STklos***. This package system provides new APIs to Scheme (e.g., network programming, cryptography, encoding, …) and it manages automatic package installation, deinstallation and testing.

11.1. *ScmPkg* "tutorial"

`stklos-pkg` is the command which gives access to *ScmPkg* in ***STklos***. This is the only command necessary to manage *ScmPkg* packages. For instance, this command manages a local cache of the *ScmPkg* server, permits (de)installation of *ScmPkg* packages, the test of packages, …

To start, we can synchronize our local repository with the *ScmPkg* servers. This can be done by the following command:

```
$ stklos-pkg --sync
```

This will download a description of the *ScmPkg* packages which are available. The list of these packages can be displayed with:

```
$ stklos-pkg --list
_bigloo-1.0.1 (tuning)
_bigloo-regexp-0.0.1
_bigloo-stdlib-0.0.1 (tuning)
_chicken-0.0.1 (tuning)
_chicken-net-0.0.1 (tuning)
_chicken-os-0.0.1
rfc3339-0.2.0
srfi1-0.0.1 (tuning)
$
```

Packages whose name start by an underscore are packages needed for alien language support (e.g. the package "*bigloo" is necessary to run a package written in the Bigloo Scheme dialect, or the package "_chicken-net" is necessary for the packages using the network primitives of the Chicken Scheme dialect). Some packages may offer a tuning. A tuned package is a package which has been specially tuned for _**STklos*** (e.g. the generic "srfi1" package which implement SRFI-1 has been tuned to be more efficient in STklos)

To download a new package (and all its dependencies), one can simply issue the following command:

```
$ stklos-pkg --extract PKG --directory /tmp/Test
```

This command downloads the package *PKG*, its dependencies and (eventually) its tuning. It also creates a `Makefile` for compiling the package. The `--directory` option specifies where the files must be extracted in the `/tmp/Test` directory (instaed of the current directory). The generated `Makefile` offers the following main targets

- **all** : the default target
- **test:** to launch the package tests
- **install:** to install the package for the user
- **system-install:** to install the package system wide (privilegied access rights are probably needed)

To test a package, one can also use the following command:

```
\type{$} stklos-pkg --test PKG
```

This downloads the necessary files in a temporary directory, and launches the tests of the package *PKG*

To install a package, one can also use the following command:

```
$ stklos-pkg --install PKG
```

This command downloads the necessary files in a temporary directory, and installs the package *PKG*.

This completes the basis *ScmPkg* tutorial. See below for the list of all stklos-pkg options

11.2. stklos-pkg options

Here are the options supported by the stklos-pkg command

```
Usage: stklos-pkg [options] [parameter ...]
Actions
  --extract=<pkg>, -e <pkg>    Extract <pkg>. Don't install it
  --test=<pkg>, -t <pkg>       Test <pkg>.
  --install=<pkg>, -i <pkg>    Extract, compile, Install <pkg>.
  --uninstall=<pkg>            un-install package <pkg>
Repository administration
  --sync, -s                   synchronize with remote server servers
  --add=<sb>, -a <sb>          Add <sb> pkgball to the local repository
  --download=<pkg>             download <pkg>
  --fill-cache                 fill cache with available distant packages
  --clear-cache                delete packages present in cache
  --reset                      reset stklos-pkg repository. USE WITH CAUTION
  --build-sync-file=<dir>      Build a synchronization file from <dir>
Informations
  --list, -l                   list available packages
  --depends=<pkg>              Show all the dependencies of <pkg>
  --installed                   Show installed packages
Misc
  --conf-dir=<dir>, -D <dir>   Use <dir> as stklos main configuration directory
  --directory=<dir>, -C <dir>  Change to directory <dir> (extract/download)
  --verbose, -v                 be verbose (can be cumulated)
  --system-wide, -S             do a system wide (de)installation
  --version, -V                 print the version and exit
  --help, -h                    display this help
  --options                     display program options
  --cp                          INTERNAL USE ONLY. Do not use this option
```

Chapter 12. Foreign Function Interface

The ***STklos*** Foreign Function Interface (FFI for short) has been defined to allow an easy access to functions written in C without needing to build C-wrappers and, consequently, without any need to write C code. FFI is very machine dependent and ***STklos*** uses the **libffi library**, a portable Foreign Function Interface library, to give access to C written code. This library supports a large set of architectures/OS that are described on its Home Page.

Moreover, since FFI allows very low level access, it is easy to crash the ***STklos*** VM when using an external C function.

Note that the support for FFI is still minimal and that it will evolve in future versions.

12.1. External functions

The definition of an external function is done with the `define-external` special form. This form takes as arguments a typed list of parameters and accepts several options to define the name of the function in the C world, the library which defines this function, … The type of the function result and the type of its arguments are defined in Table 1. This table lists the various keywords reserved for denoting types and their equivalence between the C and the Scheme worlds.

Table 1. FFI types

Name	Corresponding C type	Corresponding Scheme type
:void	void	*none*
:char	char	Scheme character
:short	short	Scheme integer
:ushort	unsigned short	Scheme integer
:int	int	Scheme integer
:uint	unsigned int	Scheme integer
:long	long int	Scheme integer
:ulong	unsigned long int	Scheme integer
:float	float	Scheme real number
:double	double	Scheme real number
:boolean	int	boolean
:pointer	void *	Scheme pointer object or Scheme string
:string	char *	Scheme string
:obj	void *	Any Scheme object passed as is

```
(define-external name parameters option)
```

The form `define-external` binds a new procedure to `name`. The arity of this new procedure is defined by the typed list of parameters given by `parameters`. This parameters list is a list of keywords (as defined in the previous table) or couples whose first element is the name of the parameter, and the second one is a type keyword. All the types defined in the above table, except `:void`, are allowed for the parameters of a foreign function.

`Define-external` accepts several options:

- `:return-type` is used to define the type of the value returned by the foreign function. The type returned must be chosen in the types specified in the table. For instance:

  ```
  (define-external maximum(:int :int)
     :return-type :int)
  ```

  defines the foreign function maximum which takes two C integers and returns an integer result. Omitting this option default to a result type equal to `:void` (i.e. the returned value is *undefined*).

- `:entry-name` is used to specify the name of the foreign function in the C world. If this option is omitted, the entry-name is supposed to be `name`. For instance:

  ```
  (define-external minimum((a :int) (b :int))
     :return-type :int
     :entry-name  "min")
  ```

  defines the Scheme function `minimum` whose application executes the C function called `min`.

- `:library-name` is used to specify the library which contains the foreign-function. If necessary, the library is loaded before calling the C function. So,

  ```
  (define-external minimum((a :int) (b :int))
     :return-type  :int
     :entry-name   "min"
     :library-name "libminmax")
  ```

  defines a function which will execute the function `min` located in the library `libminmax.xx` (where `xx` is the suffix used for shared libraries on the running system (generally `so`)).

Hereafter, are some commented definitions of external functions:

```
(define-external isatty ((fd :int))
    :return-type :boolean)
```

```
(define-external system ((cmd :string))
  :return-type :int)

(define-external ttyname (:int)
  :return-type :string)
```

All these functions are defined in the C standard library, hence it is not necessary to specify the `:library-name` option.

- ***istty*** is declared here as a function which takes an integer and returns a boolean (in fact, the value returned by the C function ***isatty*** is an ***int***, but we ask here to the FFI system to translate this result as a boolean value in the Scheme world).
- ***system*** is a function which takes a string as parameter and returns an ***int***.
- ***ttyname*** is a function whih takes an int and returns a string. Note that in this function the name of the parameter has been omitted as within C prototypes.

If an external function receives an `:int` argument and is passed a Scheme bignum, which then doesn't fit a ***long int*** in C, the external function will signal an error. When a `:float` or `:double` argument is declared and is passed a Scheme real that requires so many bits so as to not be representable in that type, that argument will be silently taken as infinity.

```
(define-external c-abs ((fd :int))
    :entry-name "abs"
    :return-type :int)

(define-external c-fabs ((fd :double))
    :entry-name "fabs"
    :return-type :double)

(define-external c-fabsf ((fd :float))
    :entry-name "fabsf"
    :return-type :float)
```

We can now use the function we have just defined:

```
(c-abs (- (expt 2 70)))    => Error
(c-fabs -1.0e+250)         => 1e+250
(c-fabsf -1.0e+250)        => +inf.0
(c-fabs (- (expt 10 300))) => 1e+300
(c-fabs (- (expt 10 600))) => +inf.0
```

In the following example we use the `:string` type. C functions accepting pointers to null-terminated strings are directly translated to this type.

The POSIX function `strpbrk` accepts two string arguments (in C, two pointers to `char`). The C call `strpbrk(str1, str2)` returns a pointer to the first occurrence in the string `str1` of one of the bytes in the string `str2`.

```
(define-external c-strpbrk ((str :string) (accept :string))
    :entry-name "strpbrk"
    :return-type :string)
```

```
(c-strpbrk "a string" "rz") => "ring"
```

Note that it would be possible to use a `:pointer` type instead for the return value, although in this case it would be more cumbersome (but does help understand the FFI better!):

```
(define-external c-strpbrk ((str :string) (accept :string))
    :entry-name "strpbrk"
    :return-type :pointer)
```

```
(c-strpbrk "a string" "rz")
                            => #[C-pointer 7f17baf94d06 @ 7f17bafb4360]
(cpointer->string (c-strpbrk "a string" "rz"))
                            => "ring"
```

Functions on C pointers are described in the next section.

12.2. C pointers

It is very common that external functions return pointers, serving as handles on internal structures. This pointers, called hereafter **cpointers**, are then boxed in a Scheme objects. This section presents the functions that can be used to deal with C pointers.

Note that by using **cpointers** objects, one gives up the safety of the Scheme environment, and care must be taken to avoid memory corruptions, errors, crashes...

STklos procedure

```
(cpointer? obj)
```

Returns `#t` is `obj` is a cpointer (a Scheme object which encapsulate a pointer to a C object), and `#f` otherwise.

STklos procedure

```
(cpointer-null? obj)
```

Returns #t is obj is a cpointer and its value is the C NULL value. Returnd #f otherwise.

STklos procedure

```
(cpointer-data obj)
(cpointer-data-set! obj adr)
```

cpointer-data returns the value associated to cpointer obj (that is the value of the pointer itself: an address).

cpointer-data-set! permits to change the pointer stored in the obj cpointer to adr. This is of course very dangerous and could lead to fatal errors.

STklos procedure

```
(cpointer-type obj)
(cpointer-type-set! obj tag)
```

cpointer-type returns the tag type associated to a cpointer. The C runtime or an extension can associate * a tag to a cpointer to make some controls (for instance, verify that obj is a cpointer on a widget structure). This function returns **void** if a type has not been set before. The semantic associated to this tag is completely left to the extension writer.

cpointer-type-set! permits to set the tag of the obj cpointer to tag (which can be of any type).

STklos procedure

```
(cpointer->string str)
```

Returns the C (null terminated) string str as a Scheme string. If str doesn't contain a C string, the result will probably result in a fatal error.

```
(define-external c-ghn ((s :pointer) (size :int))
                 :entry-name "gethostname"
                 :return-type :int)
(define name (allocate-bytes 10))

name                    => #[C-pointer 7fd830820f80 @ 7fd8305bee40]
(c-ghn name 9)          => 0
```

```
(cpointer->string name) => "socrates"
```

STklos procedure

```
(allocate-bytes n)
```

`Allocate-bytes` will allocate `n` consecutive bytes using the standard STklos allocation function (which uses the Boehm–Demers–Weiser garbage collector [BoehmGC]). It returns a `cpointer` Scheme object that points to the first byte allocated. This pointer is managed by the standard GC and doesn't need to be freed.

STklos procedure

```
(free-bytes obj)
```

`Obj` must be a `cpointer` to allocated data. When `Free-bytes` is called on `obj`, it will deallocate its data calling - the C function `free` (if it was allocated by the standard C `malloc` function), or - the Boehm GC free function (if the pointer was allocated using `allocate-bytes` primitive).

```
(define a (allocate-bytes 10))
a   => #[C-pointer 7fd91e8e0f80 @ 7fd91e897b70]
(cpointer-type-set! a 'myadress)
a   => #[myadress-pointer 7fd91e8e0f80 @ 7fd91e897b70]
(free-bytes a)
a   => #[myadress-pointer 0 @ 7fd91e897b70]
```

After the call to `free-bytes`, when `a` is printed, the first number shown is zero, indicating that its data pointer does not point to allocated memory (a NULL value for C).

Chapter 13. Using the SLIB package

SLIB is a library for the programming language Scheme, written by Aubrey Jaffer [SLIB]. It provides a platform independent framework for using packages of Scheme procedures and syntax. It uses only standard Scheme syntax and thus works on many different Scheme implementations.

To use this package, you have just to type

```
(require "slib")
```

or use the **SRFI-96** (*SLIB Prerequisites*) with

```
(import (srfi 96))
```

and follow the instructions given in the ***SLIB*** library manual to use a particular package.

> ***SLIB*** uses also the *require* and *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.
>
> ***SLIB*** needs to create a catalog of the file that must be loaded to implement a given feature. This catalog is stored in a file named `slibcat` This file is, by default located in the `~/.stklos/slib` directory. It is possible to change this directory with the `STKLOS_IMPLEMENTATION_PATH` shell variable.
>
> ***STklos*** searches the ***SLIB*** implementation directory in some standard places. If not found, you can fix it with the `SCHEME_LIBRARY_PATH` shell variable.

Chapter 14. SRFIs

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.

14.1. Supported SRFIs

STklos supports **118** finalized SRFIS. Some of these SRFIS are *embedded* and some are *external*.

An *embedded* SRFI can be directly used without any particular action, whereas an *external* needs to be loaded before use.

The following SRFIS are implemented:

- **SRFI-0** — *Feature-based conditional expansion construct*
- **SRFI-1** — *List Library*
- **SRFI-2** — *AND-LET*: an AND with local bindings, a guarded LET* special form*
- **SRFI-4** — *Homogeneous numeric vector datatypes*
- **SRFI-5** — *A compatible let form with signatures and rest arguments*
- **SRFI-6** — *Basic String Ports*
- **SRFI-7** — *Feature-based program configuration language*
- **SRFI-8** — *Receive: Binding to multiple values*
- **SRFI-9** — *Defining Record Types*
- **SRFI-10** — *Sharp Comma External Form*
- **SRFI-11** — *Syntax for receiving multiple values*
- **SRFI-13** — *String Library*
- **SRFI-14** — *Character-Set Library*
- **SRFI-15** — *Syntax for dynamic scoping (withdrawn)*
- **SRFI-16** — *Syntax for procedures of variable arity*
- **SRFI-17** — *Generalized set!*
- **SRFI-18** — *Multithreading support*
- **SRFI-19** — *Time Data Types and Procedures*
- **SRFI-22** — *Running Scheme Scripts on Unix*
- **SRFI-23** — *Error reporting mechanism*
- **SRFI-25** — *Multi-dimensional Arrays*
- **SRFI-26** — *Notation for Specializing Parameters without Currying*
- **SRFI-27** — *Source of random bits*
- **SRFI-28** — *Basic Format Strings*
- **SRFI-29** — *Localization*
- **SRFI-30** — *Nested Multi-line Comments*
- **SRFI-31** — *A special form for recursive evaluation*
- **SRFI-34** — *Exception Handling for Programs*
- **SRFI-35** — *Conditions*
- **SRFI-36** — *I/O Conditions*
- **SRFI-37** — *args-fold: a program argument processor*
- **SRFI-38** — *External representation of shared structures*

- **SRFI-39** — *Parameters objects*
- **SRFI-41** — *Streams*
- **SRFI-43** — *Vector library*
- **SRFI-45** — *Primitives for Expressing Iterative Lazy Algorithms*
- **SRFI-48** — *Intermediate Format Strings*
- **SRFI-51** — *Handling rest list*
- **SRFI-54** — *Formatting*
- **SRFI-55** — *Require-extension*
- **SRFI-59** — *Vicinity*
- **SRFI-60** — *Integers as bits*
- **SRFI-61** — *A more general COND clause*
- **SRFI-62** — *S-expression comments*
- **SRFI-64** — *A Scheme API for test suites*
- **SRFI-66** — *Octet Vectors*
- **SRFI-69** — *Basic Hash Tables*
- **SRFI-70** — *Numbers*
- **SRFI-74** — *Octet-Addressed Binary Blocks*
- **SRFI-87** — ⇒ *in case clauses*
- **SRFI-88** — *Keyword Objects*
- **SRFI-89** — *Optional Positional and Named Parameters*
- **SRFI-94** — *Type-Restricted Numerical Functions*
- **SRFI-95** — *Sorting and Merging*
- **SRFI-96** — *SLIB Prerequisites*
- **SRFI-98** — *Interface to access environment variables*
- **SRFI-100** — *define-lambda-object*
- **SRFI-111** — *Boxes*
- **SRFI-112** — *Environment Inquiry*
- **SRFI-113** — *Sets and Bags*
- **SRFI-116** — *Immutable List Library*
- **SRFI-117** — *Queues based on lists*
- **SRFI-118** — *Simple adjustable-size strings*
- **SRFI-125** — *Intermediate hash tables*
- **SRFI-127** — *Lazy Sequences*
- **SRFI-128** — *Comparators (reduced)*
- **SRFI-129** — *Titlecase procedures*
- **SRFI-130** — *Cursor-based string library*
- **SRFI-132** — *Sort Libraries*
- **SRFI-133** — *Vector Library (R7RS-compatible)*
- **SRFI-134** — *Immutable Deques*
- **SRFI-135** — *Immutable Texts*
- **SRFI-137** — *Minimal Unique Types*
- **SRFI-138** — *Compiling Scheme programs to executables*
- **SRFI-141** — *Integer Division*
- **SRFI-143** — *Fixnums*
- **SRFI-144** — *Flonums*
- **SRFI-145** — *Assumptions*
- **SRFI-151** — *Bitwise Operations*
- **SRFI-152** — *String Library (reduced)*
- **SRFI-154** — *First-class dynamic extents*
- **SRFI-156** — *Syntactic combiners for binary predicates*
- **SRFI-158** — *Generators and Accumulators*
- **SRFI-160** — *Homogeneous numeric vector libraries*
- **SRFI-161** — *Unifiable Boxes*
- **SRFI-162** — *Comparators sublibrary*
- **SRFI-169** — *Underscores in numbers*
- **SRFI-170** — *POSIX API*
- **SRFI-171** — *Transducers*
- **SRFI-173** — *Hooks*
- **SRFI-174** — *POSIX Timespecs*
- **SRFI-175** — *ASCII character library*

- **SRFI-176** — *Version flag*
- **SRFI-180** — *JSON*
- **SRFI-185** — *Linear adjustable-length strings*
- **SRFI-189** — *Maybe and Either: optional container types*
- **SRFI-190** — *Coroutines Generators*
- **SRFI-192** — *Port Positioning*
- **SRFI-193** — *Command line*
- **SRFI-195** — *Multiple-value boxes*
- **SRFI-196** — *Range Objects*
- **SRFI-207** — *String-notated bytevectors*
- **SRFI-208** — *NaN procedures*
- **SRFI-214** — *Flexvectors*
- **SRFI-215** — *Central Log Exchange*
- **SRFI-216** — *SICP Prerequisites (Portable)*
- **SRFI-217** — *Integer Sets*
- **SRFI-219** — *Define higher-order lambda*
- **SRFI-221** — *Generator/accumulator sub-library*
- **SRFI-223** — *Generalized binary search procedures*
- **SRFI-224** — *Integer Mappings*
- **SRFI-228** — *A further comparator library*
- **SRFI-229** — *Tagged Procedures*
- **SRFI-230** — *Atomic Operations*
- **SRFI-233** — *INI files*
- **SRFI-236** — *Evaluating expressions in an unspecified order*
- **SRFI-238** — *Codesets*
- **SRFI-244** — *Multiple-value definitions*

14.2. Using a SRFI

Using a particular SRFI can be done with the special form `cond-expand` defined in **SRFI-0** which is fully supported by ***STklos***. This form accepts 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***).

For instance, to use ***srfi-n***, you can use

```
(cond-expand
 (srfi-n))
```

This forms does nothing if ***srfi-n*** is an embedded SRFI and ensures that all the files needed by this SRFI will be properly loaded if it is an external SRFI.

STklos also offers the primitive **require-feature** which ensures (eventually) the loading of files needed to use a given SRFI. This primitive accepts several forms to ensure that the SRFI can be used. For instance, to use **SRFI-1** (*List Library*), the following forms are possible:

```
(require-feature 'srfi-1)
(require-feature "srfi-1")
(require-feature 1)
(require-feature 'lists)   ;; Since this feature name is an alias for SRFI-1
```

The list of the aliases defined for the supported SRFIs is given in Table 2.

14.2.1. Embedded SRFIs

As said before, an *embedded* SRFI can be used directly without loading a support file. (Note that using **require-feature** works too and permits to ignore if the SRFI is embedded).

List of embedded SRFIs: `srfi-0` `srfi-6` `srfi-8` `srfi-10` `srfi-11` `srfi-15` `srfi-16` `srfi-18` `srfi-22` `srfi-23` `srfi-28` `srfi-30` `srfi-31` `srfi-34` `srfi-38` `srfi-39` `srfi-45` `srfi-55` `srfi-62` `srfi-70` `srfi-87` `srfi-88` `srfi-98` `srfi-111` `srfi-112` `srfi-118` `srfi-138` `srfi-143` `srfi-145` `srfi-169` `srfi-176` `srfi-192` `srfi-193` `srfi-195` `srfi-208` `srfi-219` `srfi-244`

14.2.2. External SRFIs

An external SRFI needs to load at least one external file. This can be done with **require** or **require-feature**. As with embedded SRFIS, using **require-feature** permits to ignore if the SRFI is external.

List of external SRFIs: `srfi-1` `srfi-2` `srfi-4` `srfi-5` `srfi-7` `srfi-9` `srfi-13` `srfi-14` `srfi-17` `srfi-19` `srfi-25` `srfi-26` `srfi-27` `srfi-29` `srfi-35` `srfi-36` `srfi-37` `srfi-41` `srfi-43` `srfi-48` `srfi-51` `srfi-54` `srfi-59` `srfi-60` `srfi-61` `srfi-64` `srfi-66` `srfi-69` `srfi-74` `srfi-89` `srfi-94` `srfi-95` `srfi-96` `srfi-100` `srfi-113` `srfi-116` `srfi-117` `srfi-125` `srfi-127` `srfi-128` `srfi-129` `srfi-130` `srfi-132` `srfi-133` `srfi-134` `srfi-135` `srfi-137` `srfi-141` `srfi-144` `srfi-151` `srfi-152` `srfi-154` `srfi-156` `srfi-158` `srfi-160` `srfi-161` `srfi-162` `srfi-170` `srfi-171` `srfi-173` `srfi-174` `srfi-175` `srfi-180` `srfi-185` `srfi-189` `srfi-190` `srfi-196` `srfi-207` `srfi-214` `srfi-215` `srfi-216` `srfi-217` `srfi-221` `srfi-223` `srfi-224` `srfi-228` `srfi-229` `srfi-230` `srfi-233` `srfi-236` `srfi-238`

14.2.3. SRFI features

For some SRFIs, ***STklos*** accepts that uses them with a name. This names are given Table 2.

Table 2. Feature identifiers

symbol	require SRFI(s)
lists	srfi-1
and-let*	srfi-2
hvectors	srfi-4
program	srfi-7
records	srfi-9
case-lambda	srfi-16
error	srfi-23
random	srfi-27
args-fold	srfi-37
parameters	srfi-39
streams	srfi-41
rest-list	srfi-51
formatting	srfi-54

symbol	require SRFI(s)
testing	srfi-64
hash-tables	srfi-69
boxes	srfi-111
sets-bags	srfi-113
immutable-lists	srfi-116
queues-as-lists	srfi-117
adjustable-strings	srfi-118
hash-table	srfi-125
lazy-sequences	srfi-127
comparators-reduced	srfi-128
titlecase	srfi-129
sort	srfi-132
vector	srfi-133
immutable-deques	srfi-134
immutable-texts	srfi-135
integer-division	srfi-141
bitwise-ops	srfi-151
posix	srfi-170
transducers	srfi-171
hooks	srfi-173
posix-timespecs	srfi-174
ascii	srfi-175
JSON	srfi-180
maybe-either	srfi-189
ini-files	srfi-233
conditions	srfi-35 srfi-36
generators	srfi-158 srfi-190

14.3. Misc. Information

Previous section described the general way to use the SRFIS implemented in ***STklos***. This section concentrates on information not given above.

srfi-0 — Feature-based conditional expansion construct

SRFI-0 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*).

STklos `cond-expand` accepts also some feature identifiers which are the same that the ones defined in Table 2, such as *case_lambda* or *generators*.

Furthermore, the feature identifier ***stklos*** and ***STklos*** are defined for applications which need to know on which Scheme implementation they are running on.

srfi-4 — Homogeneous numeric vector datatypes

SRFI-4 is fully supported and is extended to provide the additional **c64vector** and **c128vector** types of **SRFI-160** (*Homogeneous numeric vector libraries*).

srfi-10 — Sharp Comma External Form

SRFI-10 is fully supported. This SRFI extends the STklos reader with the `#,` notation which is fully described in this document (see *primitive* `define-reader-ctor`).

srfi-16 — Syntax for procedures of variable arity

SRFI-16 is fully supported. Note that `case-lambda` is now defined in R^{7}RS.

srfi-17 — Generalized set!

SRFI-17 is fully supported. See the documentation of procedures `set!` and `setter`. However, requiring explicitly `srfi-17` permits to define the setters for the (numerous) `cXXXXr` list procedures.

srfi-19 — Time Data Types and Procedures

SRFI-19 is fully supported. STklos offers, as an extension, the procedures `date=?`, `date<?`, `date>?`, `date⇐?` and `date>=?`. These will compare dates by first normalizing them to make the time zone offset irrelevant, so "2000 Nov 12 03:30:10 GMT-2" will be taken as equal to "2000 Nov 12 02:30:10 GMT-1".

srfi-22 — Running Scheme Scripts on Unix

SRFI-22 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 (*Running Scheme Scripts on Unix*) recommends to invoke the Scheme script interpreter from the script via a /usr/bin/env trampoline, like this:

```
#!/usr/bin/env stklos
```

Here is an example of the classical `echo` command (without option) in Scheme:

```
#!/usr/bin/env stklos

(define (main arguments)
  (for-each (lambda (x) (display x) (display #\space))
            (cdr arguments))
  (newline)
  0)
```

srfi-23 — Error reporting mechanism

SRFI-23 is fully supported. Note that the ***STklos* error** is more general than the one defined in SRFI-23.

srfi-25 — Multi-dimensional Arrays

STklos implements the arrays of **SRFI-25**. All the forms defined in the SRFI are implemented in ***STklos***, but some other functions, not present in the SRFI, are documented here.

STklos procedure

```
(shape? obj)
```

Checks if `obj` is an array shape. SRFI-25 dictates that a shape is an ordinary array, with rank two and shape `(0 r 0 2)`, where `r` is the rank of the array that the shape describes. So, any array of shape `(0 r 0 2` is a shape, for any non-negative integer `r`.

STklos procedure

```
(shared-array? array)
```

Will return `#t` when the array has its data shared with other arrays, and `#f` otherwise.

STklos procedure

```
(shape-for-each shape proc [index-object])
```

This procedure will apply proc to all valid sequences of indices in `shape`, in row-major order.

If `index-object` is not provided, then `proc` must accept as many arguments as the number of dimensions that the shape describes.

```
(shape-for-each (shape 1 3 10 12)
                (lambda (x y)
                  (format #t "[~a ~a]~%" x y)))
        |- [1 10]
           [1 11]
           [2 10]
           [2 11]
```

If `index-object` is provided, it is used as a place to store the indices, so proc must accept either a vector or an array (this is to avoid pushing and popping too many values when calling `proc`). `index-object`, when present, must be aither a vector or array.

```
(let ((vec (make-vector 2 #f)))
  (shape-for-each (shape 1 3 10 12)
                  (lambda (o)
                    (format #t "[~a ~a]~%"
                    (vector-ref o 0)
                    (vector-ref o 1)))
                  vec))
        |- [1 10]
           [1 11]
           [2 10]
           [2 11]

(let ((arr (make-array (shape 0 2))))
  (shape-for-each (shape 1 3 10 12)
                  (lambda (o)
                    (format #t "[~a ~a]~%"
                    (array-ref o 0)
                    (array-ref o 1)))
                  arr))
         |- [1 10]
            [1 11]
            [2 10]
            [2 11]
```

STklos procedure

```
(share-nths a d n)
```

`Share-nths` takes every `` n`th slice along dimension `d `` into a shared array. This preserves the origin.

```
(define a (array (shape 0 4 0 4)
                 -1 -2 -3 -4
                 -5 -6 -7 -8
                 -9 -10 -11 -12
                 -13 -14 -15 -16))

(share-nths a 0 2)
 => #,(<array> (0 2 0 4) -1  -2  -3  -4
                         -9 -10 -11 -12)

(share-nths a 1 2)
 => #,(<array> (0 4 0 2) -1  -3  -5  -7
                         -9 -11 -13 -15)
```

STklos procedure

```
(share-column arr k)
```

Shares whatever the second index is about. The result has one dimension less.

```
(define a (array (shape 0 2 0 2 0 2) -1 -2 -3 -4 -5 -6 -7 -8))

(share-column a 1) => #,(<array> (0 2 0 2) -3 -4 -7 -8)
(share-column a 0) => #,(<array> (0 2 0 2) -1 -2 -5 -6)
```

STklos procedure

```
(share-row arr k)
```

Shares whatever the first index is about. The result has one dimension less.

```
(define a (array (shape 0 2 0 2 0 2) -1 -2 -3 -4 -5 -6 -7 -8))

(share-row a 0) => #,(<array> (0 2 0 2) -1 -2 -3 -4)
(share-row a 1) => #,(<array> (0 2 0 2) -5 -6 -7 -8)
```

STklos procedure

```
(share-array/origin arr k …)
```

```
(share-array/origin arr index)
```

change the origin of `arr` to `k` ..., with `index` a vector or zero-based one-dimensional array that contains k ...

```
(define a (array (shape 0 2 0 2 ) -1 -2 -3 -4))

(share-array/origin  a 1 1) => #,(<array> (1 3 1 3) -1 -2 -3 -4)
```

STklos procedure

```
(array-copy+share array)
```

Returns a copy of `array`. If array does not have its own internal data, but was built using share-array, then the new array will be similar — it will be a copy of array, sharing the elements in the same way.

STklos procedure

```
(array-size array)
```

Returns the number of elements in `array`.

STklos procedure

```
(array-shape array)
```

Returns the shape of `array`.

STklos procedure

```
(array→list array)
```

Returns a list that contains a copy of the elements of `array`, in row-major order. This is not recursive, and will not flatten the array.

STklos procedure

```
(array->vector array)
```

Returns a vector that contains a copy of the elements of `array`, in row-major order. The new vector does not share elements with the original array (it is a fresh copy). This is not recursive, and will not flatten the array.

STklos procedure

```
(array-length array dim)
```

Returns the length of dimension `dim` in array `array`.

STklos procedure

```
(array-map [shape] proc arr₀ arr₁ …)
```

This procedure is similar to `map` for lists: it will run `proc` on an element of each of the `arr`$_0$, `arr`$_1$, … arguments, storing the result in the equivalent position of a newly created array.

The shapes of the arrays must be the same.

The procedure will create a new array with shape `shape` (or `arr`$_0$'s shape, if `shape` was not specified).

STklos procedure

```
(array-map! array [shape] proc arr₀ arr₁ …)
```

For each valid index `idx`, applies proc to the corresponding position in `arr`$_0$, `arr`$_1$, … and then sets the same place in `array` to the result.

If `shape` is specified, it should specify a subarray of `array`, and only that section will be mapped.

STklos procedure

```
(array-append dim arr₁ arr₂ …)
```

Appends arrays `arr`$_1$, `arr`$_2$, … along the specified dimension `dim`. The arrays must have equally many dimensions and all other dimensions equally long.

```
(define a (array (shape 0 2 0 3) 11 22 33 44 55 66))
(define b (array (shape 0 3 0 3) -11 -22 -33 -44 -55 -66 -77 -88 -99))
(define c (array (shape 0 1 0 3) 'a 'b 'c))

(array-append 0 a b c) =>  #,(<array> (0 6 0 3)
                                        11  22  33
                                        44  55  66
                                       -11 -22 -33
                                       -44 -55 -66
                                       -77 -88 -99
                                         a   b   c)
```

STklos procedure

```
(array-share-count array)
```

Returns the number of arrays that were built sharing `array`'s elements through `(share-array array shape proc)`, and that were not yet garbage collected. Note that it may take a long time for an object to be garbage collected automatically. It is possible to force a garbage collection pass by calling `(gc)`, but even that does not guarantee that a specific object will be collected.

STklos procedure

```
(array-copy array)
```

Returns a copy of `array`. The new copy will have no data shared with any other array, even if the argument `array` did.

STklos procedure

```
(array-for-each-index arr proc [index-object])
```

Will loop through all valid indices of `array`, applying `proc` to those indices.

If `index-object` is not provided, then `proc` must accept as many arguments as the number of dimensions that the shape describes.

If `index-object` is provided, it is used as a place to store the indices, so `proc` must accept a vector or an array (this is to avoid pushing and popping too many values when calling proc). `index-object`, when present, must be aither a vector or array.

See the documentation of `shape-for-each` for more information on `index-object`.

STklos procedure

```
(tabulate-array shape proc)
(tabulate-array shape proc idx)
```

Returns a new array of shape `shape`, populated according to `proc`. Each valid index in `shape` is passed to `proc`, and the result is place in the according array position.

`idx` is an object that may be used to store the indices, and it may be either a vector or an array. If it is not present, or if it is `#f`, then an index vector will be created internally.

STklos procedure

```
(array-retabulate! arr shp proc [index-object])
```

Sets the elements of `arr` in `shape` to the value of `proc` at that index, using `index-object` if provided. This is similar to `tabulate-array!`, except that the array is given by the user.

```
(define arr (array (shape 0 2 0 2) 'a 'b 'c 'd))
(array-retabulate! arr (shape 0 2 0 2) (lambda (x y) (+ 1 x y)))
arr => #,(<array> (0 2 0 2) 1 2 2 3)
```

STklos procedure

```
(transpose arr k ...)
```

Shares `arr` with permuted dimensions. Each dimension from 0 inclusive to rank exclusive must appear once in `k` ...

This is a generalized transpose. It can permute the dimensions any which way. The permutation is provided by a permutation matrix: a square matrix of zeros and ones, with exactly one one in each row and column, or a permutation of the rows of an identity matrix; the size of the matrix must match the number of dimensions of the array.

The default permutation is `[ 0 1 , 1 0 ]` of course, but any permutation array can be specified, and the shape array of the original array is then multiplied with it, and index column vectors of the new array with its inverse, from left, to permute the rows appropriately.

```
(transpose (array (shape 0 4 0 4)
                  -1  -2   -3  -4
                  -5  -6   -7  -8
                  -9  -10 -11 -12
                  -13 -14 -15 -16))
 => #,(<array> (0 4 0 4)
              -1 -5  -9 -13
              -2 -6 -10 -14
              -3 -7 -11 -15
              -4 -8 -12 -16)

(transpose (array (shape 0 3 0 3 0 2)
                  -1 -2
                  -3 -4
                  -5 -6

                  -7 -8
                  -9 -10
                  -11 -12

                  -13 -14
                  -15 -16
                  -17 -18))
 => #,(<array> (0 2 0 3 0 3)
               -1  -7 -13
               -3  -9 -15
               -5 -11 -17

               -2  -8 -14
               -4 -10 -16
               -6 -12 -18)
```

srfi-27 — Source of random bits

SRFI-27 is fully supported. Using primitives `random-integer` or `random-real` automatically load this SRFI.

srfi-28 — Basic Format Strings

SRFI-28 is fully supported. Note that ***STklos*** `format` is more general than the one defined this SRFI.

srfi-35 — Conditions

SRFI-35 is fully supported. See *Section 7.3* for the predefined conditions and when it is required to load this file.

srfi-36 — I/O Conditions

SRFI-36 is fully supported. See *Section 7.3* Conditions) for the predefined conditions and when it is required to load this file.

srfi-55 — Require-extension

SRFI-55 is fully supported. Furthermore, ***STklos*** also accepts the symbols defined in Table 2 in a *require-extension* clause.

srfi-69 — Basic Hash Tables

SRFI-69 is fully supported. Note that the default comparison function in ***STklos*** is `eq?` whereas it is `equal?` for the SRFI. Furthermore the hash functions defined in the SRFI are not defined by default in ***STklos***. To have a fully compliant SRFI-69 behaviour, you need use a `require-feature` in your code.

srfi-88 — Keyword Objects

SRFI-88 is fully supported. The only difference between the keywords defined in the SRFI document and the ***STklos*** keywords is on the zero-length keyword: For ***STklos***, `:` is equivalent to the keyword `#:||`, whereas the SRFI considers that `:` is not a keyword but a symbol.

To obtain the symbol `:` in ***STklos***, you must use `|:|`.

srfi-116 — Immutable List Library

STklos implements the arrays of **SRFI-116**.

STklos procedure

```
(ipair a d)
```

Returns a newly allocated `ipair` whose `icar` is `a` and whose `icdr` is `d`. The ipair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

STklos procedure

```
(ilist obj …)
```

Returns a newly allocated ilist of its arguments.

```
(ilist 'a (+ 3 4) 'c)          =>  (a 7 c)
(ilist)                        =>  ()
```

Being an ilist, its CAR, CDR and all sublists are immutable.

STklos procedure

```
(xipair d a)
```

The same as `(lambda (d a) (ipair a d))`

Of utility only as a value to be conveniently passed to higher-order procedures.

```
(xipair (iq b c) 'a) => (a b c)
```

The name stands for "eXchanged Immutable PAIR."

STklos procedure

`(ipair` **obj ...)***

`ipair*` is like `ilist` except that the last argument to `ipair*` is used as the ,(emph "cdr") of the last pair constructed.

```
   (ipair* 1 2 3)         => (1 2 . 3)
   (ipair* 1 2 3 '(4 5)) => (1 2 3 4 5)
   (ipair*)               => ()
```

STklos procedure

```
(make-ilist n [fill])
```

Returns an n-element ilist, whose elements are all the value fill. If the fill argument is not given, the elements of the ilist may be arbitrary values.

```
(make-ilist 4 'c) => (c c c c)
```

STklos procedure

```
(ilist-tabulate n init-proc)
```

Returns an n-element ilist. Element i of the ilist, where 0 ⇐ i < n, is produced by `(init-proc i)`. No guarantee is made about the dynamic order in which `init-proc` is applied to these indices.

```
(ilist-tabulate 4 values) => (0 1 2 3)
```

STklos procedure

```
(ilist-copy lst)
```

Copies the spine of the argument, including the ilist tail.

STklos procedure

```
(iiota count [ start step ]
```

Returns an ilist containing the elements

`(start start+step … start+(count-1)*step)`

The `start` and `step` parameters default to 0 and 1, respectively. This procedure takes its name from the APL primitive.

```
(iiota 5) => (0 1 2 3 4)
(iiota 5 0 -0.1) => (0 -0.1 -0.2 -0.3 -0.4)
```

STklos procedure

```
(icar ipair)
(icdr ipair)
```

These procedures return the contents of the icar and icdr field of their argument, respectively. Note that it is an error to apply them to the empty ilist.

```
(icar (iq a b c))       =>  a          (icdr (iq a b c))     =>  (b c)
(icar (iq (a) b c d))   =>  (a)        (icdr (iq (a) b c d)) =>  (b c d)
(icar (ipair 1 2))      =>  1          (icdr (ipair 1 2))    =>  2
(icar '())              =>  *error*    (icdr '())            =>  *error*
```

STklos procedure

```
(ipair? obj)
```

Returns true and only if `x` is a proper ilist — that is, a ()-terminated ilist.

STklos procedure

```
(proper-ilist? x)
(ilist? x)
```

These identifiers are bound either to the same procedure. In either case, true is returned iff `x` is a proper ilist — a ()-terminated ilist.

More carefully: The empty list is a proper ilist. An ipair whose icdr is a proper ilist is also a proper ilist. Everything else is a dotted ilist. This includes non-ipair, non-() values (e.g. symbols, numbers, mutable pairs), which are considered to be dotted ilists of length 0.

STklos procedure

```
(dotted-ilist? x)
```

Returns true if `x` is a finite, non-nil-terminated ilist. That is, there exists an n >= 0 such that `icdrn(x)` is neither an ipair nor `()`. This includes non-ipair, non-() values (e.g. symbols, numbers), which are considered to be dotted ilists of length 0.

```
(dotted-ilist? x) = (not (proper-ilist? x))
```

STklos procedure

```
(not-ipair? x)
```

This is the same as `(lambda (x) (not (ipair? x)))`

Provided as a procedure as it can be useful as the termination condition for ilist-processing procedures that wish to handle all ilists, both proper and dotted.

STklos procedure

```
(null-ilist? lst)
```

Ilist is a proper ilist. This procedure returns true if the argument is the empty list `()`, and false otherwise. It is an error to pass this procedure a value which is not a proper ilist. This procedure is recommended as the termination condition for ilist-processing procedures that are not defined on dotted ilists.

STklos procedure

```
(ilist= elt= ilist1 ...)
```

Determines ilist equality, given an element-equality procedure. Proper ilist A equals proper ilist B if they are of the same length, and their corresponding elements are equal, as determined by `elt=`. If the element-comparison procedure's first argument is from ilisti, then its second argument is from ilisti+1, i.e. it is always called as `(elt= a b)` for `a` an element of ilist A, and `b` an element of ilist B.

In the n-ary case, every ilisti is compared to ilisti+1 (as opposed, for example, to comparing ilist1 to ilisti, for i>1). If there are no ilist arguments at all, `ilist=` simply returns true.

It is an error to apply ilist= to anything except proper ilists. It cannot reasonably be extended to dotted ilists, as it provides no way to specify an equality procedure for comparing the ilist terminators.

Note that the dynamic order in which the elt= procedure is applied to pairs of elements is not specified. For example, if ilist= is applied to three ilists, A, B, and C, it may first completely compare A to B, then compare B to C, or it may compare the first elements of A and B, then the first elements of B and C, then the second elements of A and B, and so forth.

The equality procedure must be consistent with `eq?`. That is, it must be the case that

```
(eq? x y) => (elt= x y)
```

Note that this implies that two ilists which are `eq?` are always ilist=, as well; implementations may exploit this fact to "short-cut" the element-by-element comparisons.

```
(ilist= eq?) => #t       ; Trivial cases
(ilist= eq? (iq a)) => #t
```

STklos procedure

```
(list-immutable+! lst)
```

```
(list-immutable! lst)
```

Destructive versions of `list→ilist`: both procedures change their argument so it will become an immutable list. `List-immutable+!` returns the list, while `list-immutable!` returns `#void`.

STklos procedure

```
(ifirst ipair)
(isecond ipair)
(ithird ipair)
(ifourth ipair)
(ififth ipair)
(isixth ipair)
(iseventh ipair)
(ieighth ipair)
(ininth ipair)
(itenth ipair)
```

Synonyms for car, cadr, caddr, …

```
(ithird '(a b c d e)) => c
```

STklos procedure

```
(icar+icdr ip)
```

The fundamental ipair deconstructor. Returns two values: the icar and the icdrif `ip`.

STklos procedure

```
(itake x i)
(idrop x i)
(ilist-tail x i)
```

`itake` returns the first i elements of ilist `x`. `idrop` returns all but the first i elements of ilist `x`. `ilist-tail` is either the same procedure as idrop or else a procedure with the same behavior.

```
 (itake (iq a b c d e)  2) => (a b)
 (idrop (iq a b c d e)  2) => (c d e)
```

`x` may be any value — a proper or dotted ilist:

```
(itake (ipair 1 (ipair 2 (ipair 3 'd)))    => (1 2)
(idrop (ipair 1 (ipair 2 (ipair 3 'd))) 2) => (3 . d)
(itake (ipair 1 (ipair 2 (ipair 3 'd))) 3) => (1 2 3)
(idrop (ipair 1 (ipair 2 (ipair 3 'd))) 3) => d
```

For a legal i, itake and idrop partition the ilist in a manner which can be inverted with iappend:

```
(iappend (itake x i) (idrop x i)) = x
```

idrop is exactly equivalent to performing i icdr operations on x; the returned value shares a common tail with x.

STklos procedure

```
(itake-right dilist i)
(idrop-right dilist i)
```

Itake-right returns the last i elements of dilist. Idrop-right returns all but the last i elements of dilist.

```
(itake-right (iq a b c d e) 2) => (d e)
(idrop-right (iq a b c d e) 2) => (a b c)
```

The returned ilist may share a common tail with the argument ilist.

dilist may be any ilist, either proper or dotted:

```
(itake-right (iq ipair 1 (ipair 2 (ipair 3 'd))) 2) => (2 3 . d)
(idrop-right (ipair 1 (ipair 2 (ipair 3 'd))) 2)    => (1)
(itake-right (ipair 1 (ipair 2 (ipair 3 'd))) 0)    => d
(idrop-right (ipair 1 (ipair 2 (ipair 3 'd))) 0)    => (1 2 3)
```

For a legal i, itake-right and idrop-right partition the ilist in a manner which can be inverted with iappend:

```
(iappend (itake dilist i) (idrop dilist i)) = dilist
```

Itake-right's return value is guaranteed to share a common tail with dilist.

STklos procedure

```
(isplit-at x i)
```

Isplit-at splits the ilist x at index i, returning an ilist of the first i elements, and the remaining tail. It is equivalent to

```
(values (itake x i) (idrop x i))
```

R^{5}RS procedure

```
(ilast ipair)
(last-ipair ipair)
```

Ilast returns the last element of the non-empty, possibly dotted, ilist ipair. Last-ipair returns the last ipair in the non-empty ilist pair.

```
(ilast (iq a b c))      => c
(last-ipair (iq a b c)) => (c)
```

STklos procedure

```
(ilength ilist)
```

Returns the length of its argument. It is an error to pass a value to ilength which is not a proper ilist (()-terminated).

The length of a proper ilist is a non-negative integer n such that icdr applied n times to the ilist produces the empty list.

STklos procedure

```
(iappend ilist1 ...)
```

Returns an ilist consisting of the elements of ilist1 followed by the elements of the other ilist parameters.

```
(iappend (iq x) (iq y))        =>  (x y)
(iappend (iq a) (iq b c d))    =>  (a b c d)
```

```
(iappend (iq a (b)) (iq (c)))  =>  (a (b) (c))
```

The resulting ilist is always newly allocated, except that it shares structure with the final ilisti argument. This last argument may be any value at all; an improper ilist results if it is not a proper ilist. All other arguments must be proper ilists.

```
(iappend (iq a b) (ipair 'c 'd))  =>  (a b c . d)
(iappend '() 'a)           =>  a
(iappend (iq x y))         =>  (x y)
(iappend)                  =>  ()
```

STklos procedure

```
(iconcatenate ilist-of-ilists)
```

Appends the elements of its argument together. That is, `iconcatenate` returns the same as

```
(iapply iappend ilist-of-ilists)
```

or, equivalently,

```
(ireduce-right iappend '() ilist-of-ilists)
```

As with `iappend`, the last element of the input list may be any value at all.

STklos procedure

```
(ireverse ilist)
```

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

```
(ireverse (iq a b c))              =>  (c b a)
(ireverse (iq a (b c) d (e (f))))  =>  ((e (f)) d (b c) a)
```

STklos procedure

```
(iappend-reverse rev-head tail)
```

`Iappend-reverse` returns `(iappend (ireverse rev-head) tail)`. It is provided because it is a common operation — a common list-processing style calls for this exact operation to transfer values accumulated in reverse order onto the front of another ilist, and because the implementation is significantly more efficient than the simple composition it replaces. (But note that this pattern of iterative computation followed by a reverse can frequently be rewritten as a recursion, dispensing with the reverse and iappend-reverse steps, and shifting temporary, intermediate storage from the heap to the stack, which is typically a win for reasons of cache locality and eager storage reclamation.)

STklos procedure

```
(izip ilist1 ilist2 ⋯)
```

Returns the same as `(lambda ilists (iapply imap ilist ilists))`

If `izip` is passed n ilists, it returns an ilist as long as the shortest of these ilists, each element of which is an n-element ilist comprised of the corresponding elements from the parameter ilists.

```
(izip (iq one two three)
     (iq 1 2 3)
     (iq odd even odd even odd even odd even))
    => ((one 1 odd) (two 2 even) (three 3 odd))

(izip (iq 1 2 3)) => ((1) (2) (3))
```

STklos procedure

```
(iunzip1 ilist)
(iunzip2 ilist)
(iunzip3 ilist)
(iunzip4 ilist)
(iunzip5 ilist)
```

`Iunzip1` takes an ilist of ilists, where every ilist must contain at least one element, and returns an ilist containing the initial element of each such ilist. That is, it returns `(imap icar ilists)`. `Iunzip2` takes an ilist of ilists, where every ilist must contain at least two elements, and returns two values: an ilist of the first elements, and an ilist of the second elements. `Iunzip3` does the same for the first three elements of the ilists, and so forth.

```
(iunzip2 (iq (1 one) (2 two) (3 three))) =>
    (1 2 3)
    (one two three)
```

STklos procedure

```
(icount pred ilist1 ilist2 …)
```

Pred is a procedure taking as many arguments as there are ilists and returning a single value. It is applied element-wise to the elements of the ilists, and a count is tallied of the number of elements that produce a true value. This count is returned. count is "iterative" in that it is guaranteed to apply pred to the ilist elements in a left-to-right order. The counting stops when the shortest ilist expires.

```
(icount even? (iq 3 1 4 1 5 9 2 5 6))            => 3
(icount < (iq 1 2 4 8) (iq 2 4 6 8 10 12 14 16)) => 3
```

STklos procedure

```
(imap proc ilist1 ilist2 …)
```

proc is a procedure taking as many arguments as there are ilist arguments and returning a single value. imap applies proc element-wise to the elements of the ilists and returns an ilist of the results, in order. The dynamic order in which proc is applied to the elements of the ilists is unspecified.

```
(imap icadr (iq (a b) (d e) (g h))) =>  (b e h)

(imap (lambda (n) (expt n n))
     (iq 1 2 3 4 5))
    =>  (1 4 27 256 3125)

(imap + (iq 1 2 3) (iq 4 5 6)) =>  (5 7 9)

(let ((count 0))
  (imap (lambda (ignored)
         (set! count (+ count 1))
         count)
       (iq a b))) =>  (1 2) or (2 1)
```

STklos procedure

```
(ifor-each proc ilist1 ilist2 …)
```

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

```
(let ((v (make-vector 5)))
  (ifor-each (lambda (i)
               (vector-set! v i (* i i)))
             (iq 0 1 2 3 4))
  v)  =>  #(0 1 4 9 16)
```

STklos procedure

```
(ifold kons knil ilist1 ilist2 ⋯)
```

The fundamental ilist iterator.

First, consider the single ilist-parameter case. If `ilist1` = `(e1 e2 ⋯ en)`, then this procedure returns

```
  (kons en ... (kons e2 (kons e1 knil)) ... )
```

That is, it obeys the (tail) recursion

```
(ifold kons knil lis) = (ifold kons (kons (icar lis) knil) (icdr lis))
(ifold kons knil '()) = knil
```

Examples:

```
  (ifold + 0 lis)                       ; Add up the elements of LIS.
  (ifold ipair '() lis)                 ; Reverse LIS.
  (ifold ipair tail rev-head)           ; See APPEND-REVERSE.
```

```
;; How many symbols in LIS?
(ifold (lambda (x count) (if (symbol? x) (+ count 1) count))
      0
      lis)

;; Length of the longest string in LIS:
(ifold (lambda (s max-len) (max max-len (string-length s)))
      0
      lis)
```

If n ilist arguments are provided, then the kons function must take n+1 parameters: one element from each ilist, and the "seed" or fold state, which is initially knil. The fold operation terminates when the shortest ilist runs out of values:

```
(ifold ipair* '() (iq a b c) (iq 1 2 3 4 5)) => (c 3 b 2 a 1)
```

STklos procedure

```
(iunfold p f g seed [tail-gen])
```

Iunfold is best described by its basic recursion:

```
(iunfold p f g seed) =
    (if (p seed) (tail-gen seed)
        (ipair (f seed)
              (iunfold p f g (g seed))))
```

```
p        Determines when to stop unfolding.
f        Maps each seed value to the corresponding ilist element.
g        Maps each seed value to next seed value.
seed     The "state" value for the unfold.
tail-gen Creates the tail of the ilist; defaults to (lambda (x) '())
```

In other words, we use g to generate a sequence of seed values seed, g(seed), g2(seed), g3(seed), ... These seed values are mapped to ilist elements by f, producing the elements of the result ilist in a left-to-right order. P says when to stop.

Iunfold is the fundamental recursive ilist constructor, just as ifold-right is the fundamental recursive ilist consumer. While iunfold may seem a bit abstract to novice functional programmers, it can be used in a number of ways:

```
;; Ilist of squares: 1^2 ... 10^2
(iunfold (lambda (x) (> x 10))
         (lambda (x) (* x x))
         (lambda (x) (+ x 1))
         1)

(iunfold null-ilist? icar icdr lis) ; Copy a proper ilist.

;; Read current input port into an ilist of values.
(iunfold eof-object? values (lambda (x) (read)) (read))

;; Copy a possibly non-proper ilist:
```

```
(iunfold not-ipair? icar icdr lis
         values)

;; Append HEAD onto TAIL:
(iunfold null-ilist? icar icdr head
        (lambda (x) tail))
```

Interested functional programmers may enjoy noting that ifold-right and iunfold are in some sense inverses. That is, given operations `knull?`, `kar`, `kdr`, `kons`, and `knil` satisfying

```
(kons (kar x) (kdr x)) = x and (knull? knil) = #t
```

then

```
(ifold-right kons knil (iunfold knull? kar kdr x)) = x
```

and

```
(iunfold knull? kar kdr (ifold-right kons knil x)) = x.
```

This combinator sometimes is called an "anamorphism;" when an explicit tail-gen procedure is supplied, it is called an "apomorphism."

STklos procedure

```
(ipair-fold kons knil ilist1 ilist2 ···)
```

Analogous to fold, but kons is applied to successive sub-ilists of the ilists, rather than successive elements — that is, kons is applied to the ipairs making up the lists, giving this (tail) recursion:

```
(ipair-fold kons knil lis) = (let ((tail (icdr lis)))
                               (ipair-fold kons (kons lis knil) tail))
(ipair-fold kons knil '()) = knil
```

Example:

```
(ipair-fold ipair '() (iq a b c)) => ((c) (b c) (a b c))
```

STklos procedure

```
(ireduce f ridentity ilist)
```

Ireduce is a variant of ifold.

Ridentity should be a "right identity" of the procedure f — that is, for any value x acceptable to f,

```
(f x ridentity) = x
```

Ireduce has the following definition:

```
If ilist = (), return ridentity;
Otherwise, return (ifold f (icar ilist) (icdr ilist)).
```

…in other words, we compute (ifold f ridentity ilist).

Note that ridentity is used only in the empty-list case. You typically use ireduce when applying f is expensive and you'd like to avoid the extra application incurred when ifold applies f to the head of ilist and the identity value, redundantly producing the same value passed in to f. For example, if f involves searching a file directory or performing a database query, this can be significant. In general, however, ifold is useful in many contexts where ireduce is not (consider the examples given in the ifold definition — only one of the five folds uses a function with a right identity. The other four may not be performed with ireduce).

```
;; take the max of an ilist of non-negative integers.
(ireduce max 0 nums) ; i.e., (iapply max 0 nums)
```

STklos procedure

```
(ifold-right kons knil ilist1 ilist2 …)
```

The fundamental ilist recursion operator.

First, consider the single ilist-parameter case. If ilist1 = (e1 e2 … en), then this procedure returns (kons e1 (kons e2 … (kons en knil)))

That is, it obeys the recursion

```
(ifold-right kons knil lis) = (kons (icar lis) (ifold-right kons knil (icdr lis)))
(ifold-right kons knil '()) = knil
```

Examples:

```
(ifold-right ipair '() lis)           ; Copy LIS.

;; Filter the even numbers out of LIS.
(ifold-right (lambda (x l) (if (even? x) (ipair x l) l)) '() lis))
```

If n ilist arguments are provided, then the `kons` procedure must take n+1 parameters: one element from each ilist, and the "seed" or fold state, which is initially `knil`. The fold operation terminates when the shortest ilist runs out of values:

```
(ifold-right ipair* '() (iq a b c) (iq 1 2 3 4 5)) => (a 1 b 2 c 3)
```

STklos procedure

```
(iunfold-right p f g seed [tail])
```

`Iunfold-right` constructs an ilist with the following loop:

```
(let lp ((seed seed) (lis tail))
  (if (p seed) lis
      (lp (g seed)
          (ipair (f seed) lis))))
```

```
p         Determines when to stop unfolding.
f         Maps each seed value to the corresponding ilist element.
g         Maps each seed value to next seed value.
seed      The "state" value for the unfold.
tail      ilist terminator; defaults to '().
```

In other words, we use g to generate a sequence of seed values

```
seed, g(seed), g2(seed), g3(seed), ...
```

These seed values are mapped to ilist elements by `f`, producing the elements of the result ilist in a right-to-left order. P says when to stop.

`Iunfold-right` is the fundamental iterative ilist constructor, just as `ifold` is the fundamental iterative ilist consumer. While `iunfold-right` may seem a bit abstract to novice functional programmers, it can be used in a number of ways:

```
;; Ilist of squares: 1^2 ... 10^2
(iunfold-right zero?
              (lambda (x) (* x x))
              (lambda (x) (- x 1))
              10)

;; Reverse a proper ilist.
(iunfold-right null-ilist? icar icdr lis)

;; Read current input port into an ilist of values.
(iunfold-right eof-object? values (lambda (x) (read)) (read))

;; (iappend-reverse rev-head tail)
(iunfold-right null-ilist? icar icdr rev-head tail)
```

Interested functional programmers may enjoy noting that ifold and `iunfold-right` are in some sense inverses. That is, given operations `knull?`, `kar`, `kdr`, `kons`, and `knil` satisfying

```
(kons (kar x) (kdr x)) = x and (knull? knil) = #t
```

then

```
(ifold kons knil (iunfold-right knull? kar kdr x)) = x
```

and

```
(iunfold-right knull? kar kdr (ifold kons knil x)) = x.
```

STklos procedure

```
(ipair-fold-right kons knil ilist1 ilist2 …)
```

Holds the same relationship with `ifold-right` that `ipair-fold` holds with `ifold`. Obeys the recursion

```
 (ipair-fold-right kons knil lis) =
     (kons lis (ipair-fold-right kons knil (icdr lis)))
 (ipair-fold-right kons knil '()) = knil
```

Example:

```
(ipair-fold-right ipair '() (iq a b c)) => ((a b c) (b c) (c))
```

STklos procedure

```
(ireduce-right f ridentity ilist)
```

`Ireduce-right` is the `fold-right` variant of `ireduce`. It obeys the following definition:

```
(ireduce-right f ridentity '()) = ridentity
(ireduce-right f ridentity (iq e1)) = (f e1 ridentity) = e1
(ireduce-right f ridentity (iq e1 e2 ...)) =
    (f e1 (ireduce f ridentity (e2 ...)))
```

...in other words, we compute `(ifold-right f ridentity ilist)`.

```
;; Append a bunch of ilists together.
;; I.e., (iapply iappend ilist-of-ilists)
(ireduce-right iappend '() ilist-of-ilists)
```

STklos procedure

```
(iappend-map f ilist1 ilist2 …)
```

Equivalent to

```
(iapply iappend (imap f ilist1 ilist2 ...))
```

and

```
(iapply iappend (imap f ilist1 ilist2 ...))
```

Map f over the elements of the ilists, just as in the `imap` function. However, the results of the applications are appended together (using `iappend`) to make the final result.

The dynamic order in which the various applications of `f` are made is not specified.

Example:

```
(iappend-map (lambda (x) (ilist x (- x))) (iq 1 3 8))
    => (1 -1 3 -3 8 -8)
```

STklos procedure

```
(ipair-for-each f ilist1 ilist2 …)
```

Like `ifor-each`, but `f` is applied to successive sub-ilists of the argument `ilists`. That is, `f` is applied to the cells of the ilists, rather than the ilists' elements. These applications occur in left-to-right order.

```
(ipair-for-each (lambda (ipair) (display ipair) (newline)) (iq a b c)) ==>
    (a b c)
    (b c)
    (c)
```

STklos procedure

```
(ifilter-map f ilist1 ilist2 …)
```

Like `imap`, but only true values are saved.

```
(ifilter-map (lambda (x) (and (number? x) (* x x))) (iq a 1 b 3 c 7))
    => (1 9 49)
```

The dynamic order in which the various applications of `f` are made is not specified.

R^{5}RS procedure

```
(imap-in-order f ilist1 ilist2 …)
```

A variant of the `imap` procedure that guarantees to apply `f` across the elements of the `ilisti` arguments in a left-to-right order. This is useful for mapping procedures that both have side effects and return useful values.

STklos procedure

```
(ifilter pred ilist )
```

Return all the elements of `ilist` that satisfy predicate `pred`. The ilist is not disordered — elements that appear in the result ilist occur in the same order as they occur in the argument `ilist`. The returned ilist may share a common tail with the argument ilist. The dynamic order in which the various applications of pred are made is not specified.

```
(ifilter even? (iq 0 7 8 8 43 -4)) => (0 8 8 -4)
```

STklos procedure

```
(ipartition pred ilist)
```

Partitions the elements of `ilist` with predicate `pred`, and returns two values: the ilist of in-elements and the ilist of out-elements. The ilist is not disordered — elements occur in the result ilists in the same order as they occur in the argument ilist. The dynamic order in which the various applications of `pred` are made is not specified. One of the returned ilists may share a common tail with the argument ilist.

```
(ipartition symbol? (iq one 2 3 four five 6)) =>
    (one four five)
    (2 3 6)
```

STklos procedure

```
(iremove pred ilist)
```

Returns ilist without the elements that satisfy predicate pred, similar to

```
(lambda (pred ilist) (ifilter (lambda (x) (not (pred x))) ilist))
```

The ilist is not disordered — elements that appear in the result ilist occur in the same order as they occur in the argument ilist. The returned ilist may share a common tail with the argument ilist. The dynamic order in which the various applications of `pred` are made is not specified.

```
(iremove even? (iq 0 7 8 8 43 -4)) => (7 43)
```

STklos procedure

```
(imember x ilist [=])
(imemq x ilist)
(imemv x ilist)
```

These procedures return the first sub-ilist of `ilis`t whose icar is `x`, where the sub-ilists of ilist are the non-empty ilists returned by `(idrop ilist i)` for `i` less than the length of ilist. If `x` does not occur in `ilist`, then false is returned. `Imemq` uses `eq?` to compare `x` with the elements of `ilist`, while `imemv` uses `eqv?`, and `imember` uses `equal?`.

```
(imemq 'a (iq a b c))           =>  (a b c)
(imemq 'b (iq a b c))           =>  (b c)
(imemq 'a (iq b c d))           =>  #f
(imemq (list 'a)
        (ilist 'b '(a) 'c))     =>  #f
(imember (list 'a)
        (ilist 'b '(a) 'c)))    =>  ((a) c)
(imemq 101 (iq 100 101 102))    =>  *unspecified*
(imemv 101 (iq 100 101 102))    =>  (101 102)
```

The comparison procedure is used to compare the elements `ei` of `ilist` to the key `x` in this way:

```
(= x ei) ; ilist is (E1 ... En)
```

That is, the first argument is always `x`, and the second argument is one of the ilist elements. Thus one can reliably find the first element of ilist that is greater than five with `(imember 5 ilist <)`

Note that fully general ilist searching may be performed with the `ifind-tail` and `ifind` procedures, e.g.

```
(ifind-tail even? ilist) ; Find the first elt with an even key.
```

R5RS procedure

```
(ifind pred ilist)
```

Return the first element of ilist that satisfies predicate `pred`; false if no element does.

```
(ifind even? (iq 3 1 4 1 5 9)) => 4
```

Note that ifind has an ambiguity in its lookup semantics — if ifind returns false, you cannot tell (in general) if it found a false element that satisfied `pred`, or if it did not find any element at all. In many situations, this ambiguity cannot arise — either the ilist being searched is known not to contain any false elements, or the ilist is guaranteed to have an element satisfying pred. However, in cases where this ambiguity can arise, you should use `ifind-tail` instead of `ifind`, since `ifind-tail` has no such ambiguity:

```
(cond ((ifind-tail pred lis) => (lambda (ipair) ...)) ; Handle (icar ipair)
      (else ...)) ; Search failed.
```

STklos procedure

```
(ifind-tail pred ilist)
```

Return the first ipair of ilist whose icar satisfies `pred`. If no ipair does, return false.

`Ifind-tail` can be viewed as a general-predicate variant of the `imember` function.

Examples:

```
(ifind-tail even? (iq 3 1 37 -8 -5 0 0)) => (-8 -5 0 0)
(ifind-tail even? (iq 3 1 37 -5)) => #f

;; IMEMBER X LIS:
(ifind-tail (lambda (elt) (equal? x elt)) lis)
```

`Ifind-tail` is essentially `idrop-while`, where the sense of the predicate is inverted: `Ifind-tail` searches until it finds an element satisfying the predicate; `idrop-while` searches until it finds an element that doesn't satisfy the predicate.

STklos procedure

```
(iany pred ilist1 ilist2 …)
```

Applies the predicate across the ilists, returning true if the predicate returns true on any application.

If there are n ilist arguments `ilist1 … ilistn`, then `pred` must be a procedure taking n arguments and returning a boolean result.

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

Note the difference between `ifind` and `iany` — `ifind` returns the element that satisfied the predicate; iany returns the true value that the predicate produced.

Like `ievery`, `iany's name does not end with a question mark — this is to indicate that it does not return a simple boolean (true or false), but a general value.

```
(iany integer? (iq a 3 b 2.7))   => #t
(iany integer? (iq a 3.1 b 2.7)) => #f
(iany < (iq 3 1 4 1 5)
       (iq 2 7 1 8 2)) => #t
```

STklos procedure

```
(ievery pred ilist1 ilist2 …)
```

Applies the predicate across the ilists, returning true if the predicate returns true on every application.

If there are n ilist arguments `ilist1 … ilistn`, then `pred` must be a procedure taking n arguments and returning a boolean result.

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

If one of the `ilisti` has no elements, ievery simply returns true.

Like `iany`, `ievery’s name does not end with a question mark — this is to indicate that it does not return a simple boolean (true or false), but a general value.

R^{5}RS procedure

```
(ilist-index pred ilist1 ilist2 …)
```

Returns the index of the leftmost element that satisfies `pred`.

If there are n ilist arguments `ilist1 … ilistn`, then `pred` must be a function taking n arguments and returning a boolean result.

`Ilist-index` applies `pred` to the first elements of the `ilisti` parameters. If this application returns true, `ilist-index` immediately returns zero. Otherwise, it iterates, applying `pred` to the second elements of the `ilisti` parameters, then the third, and so forth. When it finds a tuple of ilist elements that cause `pred` to return true, it stops and returns the zero-based index of that position in the ilists.

The iteration stops when one of the ilists runs out of values; in this case, `ilist-index` returns false.

```
ilist-index even? (iq 3 1 4 1 5 9)) => 2
ilist-index < (iq 3 1 4 1 5 9 2 5 6) (iq 2 7 1 8 2)) => 1
ilist-index = (iq 3 1 4 1 5 9 2 5 6) (iq 2 7 1 8 2)) => #f
```

STklos procedure

```
(itake-while pred ilist)
```

Returns the longest initial prefix of ilist whose elements all satisfy the predicate `pred`.

```
(itake-while even? (iq 2 18 3 10 22 9)) => (2 18)
```

STklos procedure

```
(idrop-while pred ilist)
```

Drops the longest initial prefix of `ilist` whose elements all satisfy the predicate `pred`, and returns the rest of the ilist.

```
(idrop-while even? (iq 2 18 3 10 22 9)) => (3 10 22 9)
```

STklos procedure

```
(ispan pred ilist)
(ibreak pred ilist)
```

`Ispan` splits the ilist into the longest initial prefix whose elements all satisfy `pred`, and the remaining tail. `Ibreak` inverts the sense of the predicate: the tail commences with the first element of the input ilist that satisfies the predicate.

In other words: `ispan` finds the initial span of elements satisfying `pred`, and `ibreak` breaks the ilist at the first element satisfying `pred`.

`Ispan` is equivalent to

```
(values (itake-while pred ilist)
        (idrop-while pred ilist))
```

```
(ispan even? (iq 2 18 3 10 22 9)) =>
  (2 18)
  (3 10 22 9)

(ibreak even? (iq 3 1 4 1 5 9)) =>
  (3 1)
  (4 1 5 9)
```

STklos procedure

```
(idelete x ilist [=])
```

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

The ilist is not disordered — elements that appear in the result ilist occur in the same order as they occur in the argument ilist. The result may share a common tail with the argument ilist.

Note that fully general element deletion can be performed with the iremove procedures, e.g.:

```
;; idelete all the even elements from LIS:
(iremove even? lis)
```

The comparison procedure is used in this way: `(= x ei)`. That is, `x` is always the first argument, and an ilist element is always the second argument. The comparison procedure will be used to compare each element of ilist 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 an ilist with `(idelete 5 ilist <)`.

STklos procedure

```
(ialist-cons key datum ialist)
```

Constructs a new ialist entry mapping key → datum onto ialist. This is the same as

```
(lambda (key datum ialist) (ipair (ipair key datum) ialist))
```

STklos procedure

```
(idelete-duplicates ilist [=])
```

Idelete-duplicates removes duplicate elements from the ilist argument. If there are multiple equal elements in the argument ilist, the result ilist only contains the first or leftmost of these elements in the result. The order of these surviving elements is the same as in the original ilist — idelete-duplicates does not disorder the ilist (hence it is useful for "cleaning up" immutable association lists).

The = parameter is used to compare the elements of the ilist; it defaults to equal?. If x comes before y in ilist, then the comparison is performed (= x y). The comparison procedure will be used to compare each pair of elements in ilist no more than once; the order in which it is applied to the various pairs is not specified.

Although idelete-duplicates can be implemented so it runs in time O(n2) for n-element ilists, the STklos implementation runs in linear expected time.

```
(idelete-duplicates (iq a b a c a b c z)) => (a b c z)

;; Clean up an ialist:
(idelete-duplicates (iq (a . 3) (b . 7) (a . 9) (c . 1))
                   (lambda (x y) (eq? (icar x) (icar y))))
    => ((a . 3) (b . 7) (c . 1))
```

STklos procedure

```
(iassoc key ialist [=] → ipair or #f
(iassq key ialist → ipair or #f
(iassv key ialist → ipair or #f
```

Ialist must be an immutable association list — an ilist of ipairs. These procedures find the first ipair in ialist whose icar field is key, and returns that ipair. If no ipair in ialist has key as its icar, then false is returned. Iassq uses eq? to compare key with the icar fields of the ipairs in ialist, while iassv uses eqv? and iassoc uses equal?.

```
(define e (iq (a 1) (b 2) (c 3)))
(iassq 'a e)                               =>  (a 1)
(iassq 'b e)                               =>  (b 2)
(iassq 'd e)                               =>  #f
(iassq (ilist 'a) (iq ((a)) ((b)) ((c))))  =>  #f
(iassoc '(a) (ilist '((a)) '((b)) '((c)))) =>  ((a))
(iassq 5 (iq (2 3) (5 7) (11 13)))         =>  *unspecified*
(iassv 5 (iq (2 3) (5 7) (11 13)))         =>  (5 7)
```

The comparison procedure is used to compare the elements ei of ilist to the key parameter in this way:

```
(= key (icar ei)) ; ilist is (E1 ... En)
```

That is, the first argument is always key, and the second argument is one of the ilist elements. Thus one can reliably find the first entry of ialist whose key is greater than five with `(iassoc 5 ialist <)`.

Note that fully general ialist searching may be performed with the ifind-tail and ifind procedures, e.g.

```
;; Look up the first association in ialist with an even key:
(ifind (lambda (a) (even? (icar a))) ialist)
```

STklos procedure

```
(ialist-delete key ialist [=])
```

`Ialist-delete` deletes all associations from ialist with the given key, using key-comparison procedure `=`, which defaults to `equal?`. The dynamic order in which the various applications of `=` are made is not specified.

Return values may share common tails with the ialist argument. The ialist is not disordered — elements that appear in the result ialist occur in the same order as they occur in the argument ialist.

The comparison procedure is used to compare the element keys ki of ialist's entries to the key parameter in this way: `(= key ki)`. Thus, one can reliably remove all entries of ialist whose key is greater than five with `(ialist-delete 5 ialist <)`

STklos procedure

```
(replace-icar ipair object)
(replace-icdr ipair object)
```

`Replace-icar` returns an ipair with object in the icar field and the icdr of ipair in the icdr field.

`Replace-icdr` returns an ipair with object in the icdr field and the icar of ipair in the icar field.

STklos procedure

```
(list→ilist lst)
```

```
(ilist→list lst)
```

These procedures return an ilist and a list respectively that have the same elements as the argument. The tails of dotted (i)lists are preserved in the result, which makes the procedures not inverses when the tail of a dotted ilist is a list or vice versa. The empty list is converted to itself.

It is an error to apply list→ilist to a circular list.

STklos procedure

```
(pair→ipair pair)
(ipair→pair ipair)
```

These procedures, which are inverses, return an ipair and a pair respectively that have the same (i)car and (i)cdr fields as the argument.

STklos procedure

```
(tree→itree object)
(itree→tree object)
```

These procedures walk a tree of pairs or ipairs respectively and make a deep copy of it, returning an isomorphic tree containing ipairs or pairs respectively. The result may share structure with the argument. If the argument is not of the expected type, it is returned.

These procedures are not inverses in the general case. For example, a pair of ipairs would be converted by `tree→itree` to an ipair of ipairs, which if converted by `itree→tree` would produce a pair of pairs.

STklos procedure

```
(gtree→itree object)
(gtree→tree object)
```

These procedures walk a generalized tree consisting of pairs, ipairs, or a combination of both, and make a deep copy of it, returning an isomorphic tree containing only ipairs or pairs respectively. The result may share structure with the argument. If the argument is neither a pair nor an ipair, it is returned.

STklos procedure

```
(iapply procedure object ⋯ ilist)
```

The `iapply` procedure is an analogue of `apply` whose last argument is an ilist rather than a list. It is equivalent to

```
(apply procedure object ... (ilist->list ilist))
```

srfi-138 — Compiling Scheme programs to executables

SRFI-138 is fully supported. The `stklos-compile` program conforms to SRFI 138, accepting all the required command line options.

The -D x flag of `stklos-compile` will define a feature named `x` for use with `cond-expand` in the compiled code only. It will not include `x` in the features list of the runtime.

srfi-145 — Assumptions

SRFI-145 is fully supported. See the `assume` *special form*.

srfi-169 — Underscores in numbers

SRFI-169 is fully supported. See *parameter* `accept-srfi-169-numbers` to eventually forbid the usage of underscores in numbers.

srfi-216 — SICP Prerequisites (Portable)

SRFI-216 is fully supported. However, it defines the constant `stream-null` and the predicate `stream-null?` which are incompatible with the ones defined in the `(stream primitive)` library used by **SRFI-41** or **SRFI-221**. Prefix the imported symbols of this SRFI, if you plan to use it with one of the previous libraries.

srfi-238 — Codesets

SRFI-238 is fully supported. Furthermore, ***STklos*** adds the functions `codeset-list` and `make-codeset`.

STklos procedure

```
(codeset-list)
```

Retuns a list of known codeset names.

```
(codeset-list) => (errno signal)
```

STklos procedure

```
(make-codeset name lst)
```

returns a new codeset object of the given `name` (a symbol). The list `lst` is a list of triplets (code-number symbol message) where `symbol` and `message` can be `#f` if `code-number` has no associated name or message.

```
(define cs
    (make-codeset 'foo
                  '((1   OK    "Everything is OK")
                    (1   YES   #f)                    ; Other name for OK
                    (2   KO    "We have a problem")
                    (2   NO    #f)                    ; Other name for KO
                    (3   MAYBE "To be determined")
                    (404 #f    "Not found"))))        ; No symbolic name
```

Bibliography

- [AMOP] Gregor Kickzales, Jim de Rivières and Daniel G. Bobrow — **The Art of Meta Object Protocol** — MIT Press, 1991.
- [Bigloo] Manuel Serrano — **Bigloo** Home Page.
- [BoehmGC] Hans Boehm — A garbage collector for C and C++ Home Page.
- [CLtL2] Guy L. Steele Jr. — **Common Lisp: the Language, 2nd Edition** — Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1990.
- [DSSSL] ISO/IEC — **Information technology, Processing Languages, Document Style Semantics and Specification Languages (DSSSL)** — 10179:1996(E), ISO, , 1996.
- [Dylan] Apple Computer — **Dylan: an Object Oriented Dynamic Language** — Apple, April, 1992.
- [GTK] **The GTK+ Toolkit Home Page**
- [PCRE] Philip Hazel — **PCRE (Perl Compatible Regular Expressions)** Home page.
- [POSIX] POSIX Committee — **System Application Program Interface (API) [C Language]** — 2017.
- [QuG92] C. Queinnec and J-M. Geffroy — **Partial Evaluation Applied to Symbolic Pattern Matching with Intelligent Backtrack** — Workshop in Static Analysis, Bigre, (81—82), Bordeaux (France), September, 1992.
- [R5RS] Kelsey, R. and Clinger, W. and Rees, J. — **The Revised5 Report on the Algorithmic Language Scheme** — Higher-Order and Symbolic Computation, 11(1), Sep, 1998.
- [R7RS] Alex S. Shinn, John Cowan and Arthur A. Gleckler — **The Revised7 Report on the Algorithmic Language Scheme — R7RS small** — July, 2013.
- [SLIB] Aubrey Jaffer — **The SLIB Portable Scheme Library Home Page**
- [SOS] Chris Hanson — The SOS Reference Manual, version 1.9 — November, 2011.
- [STk] Erick Gallesio — **STk Reference Manual** — RT 95-31a, I3S CNRS / Université de Nice - Sophia Antipolis, juillet, 1995.
- [Tiny-Clos] Gregor Kickzales — **Tiny-Clos** — December, 1992.
- [Tk] John K. Ousterhout — **An X11 toolkit based on the Tcl Language** — USENIX Winter Conference, January, 1991, pp. 105—115.
- [TuD96] Sho-Huan Simon Tung and R. Kent Dybvig — **Reliable Interactive Programming with Modules** — LISP and Symbolic Computation, 91996, pp. 343—358.

Appendix A: STklos Idiosyncrasies

A.1. STklos libraries

This section describes the standard libraries provided by ***STklos***.

A.1.1. The (scheme …) Libraries

R^{7}RS Small Libraries

STklos offers all the libraries defined by R^{7}RS:

- base
- case-lambda
- char
- complex
- cxr
- eval
- file
- inexact
- lazy
- load
- process-context
- r5rs
- read
- repl
- time
- write

See the [R7RS] document for more information.

R^{7}RS Large Libraries

STklos supports some libraries of the (under development) R^{7}RS Large editions.

For now, supported libraries of the **Red edition** are

- box *(srfi 111)*
- charset *(srfi 14)*
- comparator *(srfi 128)*
- hash-table *(srfi 125)*

- ideque *(srfi 134)*
- ilist *(srfi 116)*
- list *(srfi 1)*
- list-queue *(srfi 117)*
- lseq *(srfi 127)*
- set *(srfi 113)*
- sort *(srfi 132)*
- stream *(srfi 41)*
- text *(srfi 135)*
- vector *(srfi 133)*

For now, supported libraries of the **Tangerine edition** are

- bitwise *(srfi 151)*
- division *(srfi 141)*
- flonum *(srfi 144)*
- generator *(srfi 158)*

A.1.2. The (srfi ...) Libraries

All the SRFI supported by ***STklos*** are placed under the `srfi` meta library and their name is SRFI number. Hence, to use the exported symbols of **SRFI-1**, you'll have to import the `(srfi 1)` library.

See *Chapter 14* for more information

A.1.3. The (stklos ...) Libraries

This section describes the standard libraries which are placed under the `stklos` meta library. Note that ***STklos*** extensions can add some libraries in the `stklos` meta library; they will be described in the extension documentation.

(stklos itrie) Library

This library was designed by Jerônimo Pellegrini (@jpellegrini).

Small description needed

The symbols exported by `(stklos itrie)` are described below:

STklos procedure

```
(alist→fxmapping alist) → fxmapping
```

Returns a newly allocated fxmapping containing the associations of alist. It is an error if the car of any pair in alist is not a fixnum. If an integer k appears as the key of multiple associations in alist (i.e. as the car of multiple pairs), then the first association for k is preferred.

```
(fxmapping->alist
  (alist->fxmapping '((1 . b) (0 . a) (2 . c))))
 => ((0 . a) (1 . b) (2 . c))

(fxmapping->alist
  (alist->fxmapping '((-10 . "yar") (-10 . "worf"))))
 => ((-10 . "yar"))
```

STklos procedure

```
(fxmapping k1 v1 k2 v2 … kn vn)
(constant-fxmapping k1 v1 k2 v2 … kn vn)
```

Builds a fixnum map containing the integer keys `k1`, `k2`, ..., `kn` with respective associated values `v1`, `v2`, ... `vn`.

It is an error if any of the keys is not an integer, or if the number of arguments is not even.

STklos procedure

```
(iset n1 n2 … nk)
(constant-iset n1 n2 … nk)
```

Builds a fixnum set containing the fixnums `n1`, `n2`, ..., `nk`.

It is an error if any of the keys is not an integer.

STklos procedure

```
(fxmapping-adjoin fxmap k1 obj1 k2 …)
```

Returns a fxmapping containing all of the associations of `fxmap` as well as the associations `(k1, obj1)`, `(k2, obj2)`, ... The number of key/value arguments must be even.

If any of the keys already have associations in `fxmap`, the old associations are preserved.

```
(fxmapping->alist (fxmapping-adjoin (fxmapping 1 'b) 0 'a))
```

```
 => ((0 . a) (1 . b))
```

STklos procedure

```
(fxmapping-contains? map element)
```

Returns true if `map` contains an association for `element`, and false otherwise.

STklos procedure

```
(fxmapping-empty? obj)
```

Returns `#t` is `obj` is an empty fxmapping and `#f` if it is an fxmapping containing at least one key. If `obj` is not an fxmapping object, an error is sginaled.

STklos procedure

```
(fxmapping-height trie)
```

Returns the height of the internal trie of an fxmap. The expected running time of searches and insertions is proportional to this value.

STklos procedure

```
(fxmapping-keys fxmap)
```

Returns the keys of `fxmap` as a list in ascending numerical order.

```
(fxmapping-keys (fxmapping 137 'a -24 'b -5072 'c))
 => (-5072 -24 137)
```

STklos procedure

```
(fxmapping-mutable? obj)
```

Returns #t is obj is a mutable fxmapping and #f otherwise.

STklos procedure

```
(fxmapping-ref/default map k obj)
```

If an association (k, v) occurs in map, returns v. Otherwise, returns obj.

```
(fxmapping-ref/default (fxmapping 36864 'zap) 36864 #f) => zap
(fxmapping-ref/default (fxmapping 0 'a) 36864 #f) => #f
```

STklos procedure

```
(fxmapping-size trie)
```

Returns the number of key/value pairs in an fxmap.

STklos procedure

```
(fxmapping-values fxmap)
```

Returns the values of fxmap as a list in ascending numerical order of key. That is, if (k1, v1), ⋯, (kn, vn) are the associations of fxmap ordered so that k1 ⇐ ⋯ ⇐ kn, then (fxmapping-values fxmap) produces the list (v1 ⋯ vn).

```
(fxmapping-values (fxmapping 0 "picard" 1 "riker" 2 "troi"))
 => ("picard" "riker" "troi")
```

STklos procedure

```
(fxmapping-union fxmap1 fxmap2 fxmap3 ⋯)
(fxmapping-intersection fxmap1 fxmap2 fxmap3 ⋯)
(fxmapping-difference fxmap1 fxmap2 fxmap3 ⋯)
(fxmapping-xor fxmap1 fxmap2)
```

Return a fxmapping whose set of associations is the union, intersection, asymmetric difference, or symmetric difference of the sets of associations of the fxmaps. Asymmetric difference is extended to more than two fxmappings by taking the difference between the first fxmapping and the union of the others. Symmetric difference is not extended beyond two fxmappings. When comparing associations, only the keys are compared. In case of duplicate keys, associations in the result fxmapping are drawn from the first fxmapping in which they appear.

```
(fxmapping->alist (fxmapping-union (fxmapping 0 'a 2 'c)
                                   (fxmapping 1 'b 3 'd)))
   => ((0 . a) (1 . b) (2 . c) (3 . d))

 (fxmapping->alist
  (fxmapping-intersection (fxmapping 0 'a 2 'c)
                          (fxmapping 1 'b 2 'c 3 'd)
                          (fxmapping 2 'c 4 'e)))
    => ((2 . c))

(fxmapping->alist
 (fxmapping-difference (fxmapping 0 'a 1 'b 2 'c)
                       (fxmapping 2 "worf")
                       (fxmapping 1 "data")))
    => ((0 . a))
```

STklos procedure

```
(fxmapping? obj)
```

Returns `#t` is `obj` is an fxmapping object and `#f` otherwise.

STklos procedure

```
(iset=? iset1 iset2 iset3 …)
(iset<? iset1 iset2 iset3 …)
(iset>? iset1 iset2 iset3 …)
(iset<=? iset1 iset2 iset3 …)
(iset>=? iset1 iset2 iset3 …)
```

These procedures return true when each set is equal (`iset=?`) or a proper subset (`iset<?`), a proper superset (`iset>?`), a subset (`iset<=?`) or a superset (`iset>=?`) of the next one.

```
(iset=? (iset 1 2 3) (iset 3 1 2))           => #t
(iset<? (iset 3 1 2) (iset 4 2 1 3))         => #t
(iset>=? (iset 3 0 1) (iset 0 1) (iset 0 1)) => #t
```

STklos procedure

```
(iset→list set)
```

Returns a newly allocated list containing the members of `set` in increasing numerical order.

```
(iset->list (iset 2 3 5 7 11)) => (2 3 5 7 11)
```

STklos procedure

```
(iset-adjoin set element1 element2 …)
(iset-adjoin! set element1 element2 …)
```

The `iset-adjoin` procedure returns a newly allocated iset that contains all the values of `set`, and in addition each element unless it is already equal to one of the existing or newly added members.

```
(iset->list (iset-adjoin (iset 1 3 5) 0)) => (0 1 3 5)
```

The `iset-adjoin!` procedure is the linear update version of `iset-adjoin`. In STklos, it is an alias to `iset-adjoin`.

STklos procedure

```
(iset-any? pred? set)
```

Returns true if at least one of the elements of `set` satisfies `pred?`. Note that this differs from the SRFI 1 analogue because it does not return an element of the iset.

```
(iset-any odd? (iset 10 2 -3 4))    => #t
(iset-any odd? (iset 10 2 -8 4 0))  => #f
```

STklos procedure

```
(iset-open-interval set low high)
(iset-closed-interval set low high)
(iset-open-closed-interval set low high)
(iset-closed-open-interval set low high)
```

Procedures that return a subset of `set` contained in the interval from `low` to `high`. The interval may be open, closed, open below and closed above, or open above and closed below.

```
(iset->list (iset-open-interval (iset 2 3 5 7 11) 2 7))        => (3 5)
(iset->list (iset-closed-interval (iset 2 3 5 7 11) 2 7))      => (2 3 5 7)
(iset->list (iset-open-closed-interval (iset 2 3 5 7 11) 2 7)) => (3 5 7)
(iset->list (iset-closed-open-interval (iset 2 3 5 7 11) 2 7)) => (2 3 5)
```

STklos procedure

```
(iset-contains? set element)
```

Returns true if `set` contains `element`, and false otherwise.

STklos procedure

```
(iset-copy set)
```

Returns a newly allocated iset containing the elements of `set`.

STklos procedure

```
(iset-count pred? set)
```

Returns the number of elements of `set` that satisfy `pred?` as an exact integer.

```
(iset-count odd? (iset 10 2 1 -3 9 4 3))  => 4
```

STklos procedure

```
(iset-delete set element1 element2 …)
(iset-delete! set element1 element2 …)
(iset-delete-all set element-list)
(iset-delete-all! set element-list)
```

The `iset-delete` procedure returns a newly allocated iset containing all the values of `set` except for any that are equal to one or more of the elements. Any element that is not equal to some member of the set is ignored.

The iset-delete! procedure is the same as iset-delete. is permitted to mutate and return the iset argument rather than allocating a new iset — but in STklos, it doesn't.

The iset-delete-all and iset-delete-all! procedures are the same as iset-delete and iset-delete!, except that they accept a single argument which is a list of elements to be deleted.

```
(iset->list (iset-delete (iset 1 3 5) 3)) => (1 5)
(iset->list (iset-delete-all (iset 2 3 5 7 11)
                             '(3 4 5)))   => (2 7 11)
```

STklos procedure

```
(iset-delete-min set)
(iset-delete-min! set)
(iset-delete-max set)
(iset-delete-max! set)
```

Returns two values: the smallest/largest integer n in set and a newly-allocated iset that contains all elements of set except for n. It is an error if iset is empty.

The iset-delete-min! and iset-delete-max! procedures are the same as iset-delete-min and iset-delete-max, respectively, except that they are permitted to mutate and return the set argument instead of allocating a new iset. In STklos, they do not.

```
(let-values (((n set) (iset-delete-min (iset 2 3 5 7 11))))
  (list n (iset->list set)))
  => (2 (3 5 7 11))
(let-values (((n set) (iset-delete-max (iset 2 3 5 7 11))))
  (list n (iset->list set)))
  => (11 (2 3 5 7))
```

STklos procedure

```
(iset-disjoint? iset1 iset2)
```

Returns #t if iset1 and iset2 have no elements in common and #f otherwise.

```
(iset-disjoint? (iset 1 3 5) (iset 0 2 4)) => #t
(iset-disjoint? (iset 1 3 5) (iset 2 3 4)) => #f
```

STklos procedure

```
(iset-empty? obj)
```

Returns #t is obj is an empty iset and #f if it is an iset containing at least one key. If obj is not an iset object, an error is sginaled.

STklos procedure

```
(iset-every? predicate iset)
```

Returns #t if every element of set satisfies predicate, or #f otherwise. Note that this differs from the SRFI 1 analogue because it does not return an element of the iset.

```
(iset-every? (lambda (x) (< x 5)) (iset -2 -1 1 2)) => #t
(iset-every? positive? (iset -2 -1 1 2))            => #f
```

STklos procedure

```
(iset-filter predicate set)
(iset-filter! predicate set)
```

Returns a newly allocated iset containing just the elements of set that satisfy predicate.

```
(iset->list (iset-filter (lambda (x) (< x 6)) (iset 2 3 5 7 11)))
 => (2 3 5)
```

iset-filter! is allowed to modify set, but in STklos it does not.

STklos procedure

```
(iset-find predicate set failure)
```

Returns the smallest element of set that satisfies predicate, or the result of invoking failure with no arguments if there is none.

```
(iset-find positive? (iset -1 1) (lambda () #f))  => 1
```

```
(iset-find zero?     (iset -1 1) (lambda () #f))  => #f
```

STklos procedure

```
(iset-fold proc nil set)
(iset-fold-right proc nil set)
```

Invokes `proc` on each member of `set` in increasing/decreasing numerical order, passing the result of the previous invocation as a second argument. For the first invocation, `nil` is used as the second argument. Returns the result of the last invocation, or `nil` if there was no invocation.

```
(iset-fold + 0 (iset 2 3 5 7 11))             => 28
(iset-fold cons '() (iset 2 3 5 7 11))        => (11 7 5 3 2)
(iset-fold-right cons '() (iset 2 3 5 7 11)) => (2 3 5 7 11)
```

STklos procedure

```
(iset-for-each proc set)
```

Applies `proc` to `set` in increasing numerical order, discarding the returned values. Returns an unspecified result.

```
(let ((sum 0))
  (iset-for-each (lambda (x) (set! sum (+ sum x)))
                 (iset 2 3 5 7 11))
  sum)
=> 28
```

STklos procedure

```
(iset-height trie)
```

Returns the height of the internal trie of an iset. The expected running time of searches and insertions is proportional to this value.

STklos procedure

```
(iset-map proc set)
```

Applies `proc` to each element of `set` in arbitrary order and returns a newly allocated iset, created as if by iset, which contains the results of the applications. It is an error if `proc` returns a value that is not an exact integer.

```
(iset-map (lambda (x) (* 10 x)) (iset 1 11 21))
     => (iset 10 110 210)
(iset-map (lambda (x) (quotient x 2))
         (iset 1 2 3 4 5))
 => (iset 0 1 2)
```

STklos procedure

```
(iset-min set)
(iset-max set)
```

Returns the smallest or largest integer in `set`, or `#f` if there is none.

```
(iset-min (iset 2 3 5 7 11)) => 2
(iset-max (iset 2 3 5 7 11)) => 11
(iset-max (iset))            => #f
```

STklos procedure

```
(iset-member element set default)
```

Returns the element of `set` that is equal to `element`. If `element` is not a member of `set`, then `default` is returned.

STklos procedure

```
(iset-mutable? obj)
```

Returns `#t` is `obj` is a mutable iset and `#f` otherwise.

STklos procedure

```
(iset-partition predicate set)
(iset-partition! predicate set)
```

Returns two values: a newly allocated iset that contains just the elements of `set` that satisfy `predicate` and another newly allocated iset that contains just the elements of `set` that do not satisfy `predicate`.

```
(let-values (((low high) (iset-partition (lambda (x) (< x 6))
                                         (iset 2 3 5 7 11))))
  (list (iset->list low) (iset->list high)))
 => ((2 3 5) (7 11))
```

STklos procedure

```
(iset-remove predicate set)
(iset-remove! predicate set)
```

Returns a newly allocated `set` containing just the elements of `set` that do not satisfy `predicate`.

```
(iset->list (iset-remove (lambda (x) (< x 6)) (iset 2 3 5 7 11)))
 => (7 11)
```

`Iset-remove!` is allowed to modify `set`, but in STklos it does not.

STklos procedure

```
(iset-search set element failure success)
(iset-search! iset element failure success)
```

`Set` is searched from lowest to highest value for `element`. If it is not found, then the `failure` procedure is tail-called with two continuation arguments, `insert` and `ignore`, and is expected to tail-call one of them. If `element` is found, then the `success` procedure is tail-called with the matching element of `set` and two continuations, `update` and `remove`, and is expected to tail-call one of them.

The effects of the continuations are as follows (where `obj` is any Scheme object):

Invoking `(insert obj)` causes `element` to be inserted into iset.

Invoking `(ignore obj)` causes `set` to remain unchanged.

Invoking `(update new-element obj)` causes `new-element` to be inserted into `set` in place of `element`.

Invoking `(remove obj)` causes the matching element of `set` to be removed from it.

In all cases, two values are returned: an iset and `obj`.

The `iset-search!` procedure is the same as `iset-search`, except that it is permitted to mutate and return the iset argument rather than allocating a new iset. In STklos, it does not.

STklos procedure

```
(iset-size set)
```

Returns the number of fixnums in `set`.

STklos procedure

```
(iset-unfold stop? mapper successor seed)
```

Create a newly allocated iset as if by iset. If the result of applying the predicate `stop?` to `seed` is true, return the iset. Otherwise, apply the procedure `mapper` to `seed`. The value that `mapper` returns is added to the iset. Then get a new seed by applying the procedure `successor` to `seed`, and repeat this algorithm.

```
(iset->list (iset-unfold (lambda (n) (> n 64))
                         values
                         (lambda (n) (* n 2))
                         2))
=> (2 4 8 16 32 64)
```

STklos procedure

```
(iset-union iset1 iset2 iset3 …)
(iset-intersection iset1 iset2 iset3 …)
(iset-difference iset1 iset2 iset3 …)
(iset-xor iset1 iset2)
(iset-union! iset1 iset2 iset3 …)
(iset-intersection! iset1 iset2 iset3 …)
(iset-difference! iset1 iset2 iset3 …)
(iset-xor! iset1 iset2)
```

Return a newly allocated iset that is the union, intersection, asymmetric difference, or symmetric difference of the isets. Asymmetric difference is extended to more than two isets by taking the difference between the first iset and the union of the others. Symmetric difference is not extended beyond two isets. Elements in the result iset are drawn from the first iset in which they appear.

```
(iset->list (iset-union (iset 0 1 3) (iset 0 2 4))) => (0 1 2 3 4)
(iset->list (iset-intersection (iset 0 1 3 4) (iset 0 2 4))) => (0 4)
(iset->list (iset-difference (iset 0 1 3 4) (iset 0 2) (iset 0 4))) => (1 3)
(iset->list (iset-xor (iset 0 1 3) (iset 0 2 4))) => (1 2 3 4)
```

The procedures whose name end in `!` are linear update procedures. The specification says they may or may not alter their argument. In STklos they do not: in fact, they are aliases to the pure functional versions.

STklos procedure

```
(iset? obj)
```

Returns `#t` is `obj` is an iset and `#f` otherwise.

STklos procedure

```
(isubset= set k)
(isubset< set k)
(isubset⇐ set k)
(isubset> set k)
(isubset>= set k)
```

Procedures that return an integer set containing the elements of `set` that are equal to, less than, less than or equal to, greater than, or greater than or equal to `k`. Note that the result of `isubset=` contains at most one element.

```
(iset->list (isubset= (iset 2 3 5 7 11) 7))  => (7)
(iset->list (isubset< (iset 2 3 5 7 11) 7))  => (2 3 5)
(iset->list (isubset>= (iset 2 3 5 7 11) 7)) => (7 11)
```

STklos procedure

```
(list→iset list)
(list→iset! set list)
```

Returns a newly allocated iset, created as if by iset, that contains the elements of `list`. Duplicate elements are omitted.

```
(list->iset '(-3 -1 0 2)) = (iset -3 -1 0 2)
```

`list→iset!` may mutate `set` rather than allocating a new iset, but in STklos it does not.

```
(iset->list (list->iset! (iset 2 3 5) '(-3 -1 0))) ⇒ (-3 -1 0 2 3 5)
```

STklos procedure

```
(make-range-iset start end [step])
```

Returns a newly allocated iset specified by an inclusive lower bound `start`, an exclusive upper bound `e`nd, and a step value (default 1), all of which are exact integers. This constructor produces an iset containing the sequence

`start, (+ start step), (+ start (* 2 step)), …, (+ start (* n step))`,

where `n` is the greatest integer such that `(+ start (* n step)) < end` if `step` is positive, or such that `(+ start (* n step)) > end` if `step` is negative. It is an error if `step` is zero.

```
(iset->list (make-range-iset 25 30))    => (25 26 27 28 29)
(iset->list (make-range-iset -10 10 6)) => (-10 -4 2 8)
```

(stklos preproc) Library

! This library must be described

A.2. STklos compiler flags

STklos compiler behaviour can be customized by several parameters. Those parameters are described below.

STklos procedure

```
(compiler:time-display)
(compiler:time-display bool)
```

This parameter controls if the time used for compiling a file must be displayed or not. It defaults to `#t`.

STklos procedure

```
(compiler:gen-line-number)
(compiler:gen-line-number bool)
```

This parameter controls if the compiled code must embed indications of the file location of the of the source expressions. When set, this parameter makes programs slower and bigger. However, it can be useful when debugging a program. This parameter defaults to `#f` (but is set to ``#t` when ***STklos*** is launched in debug mode).

STklos procedure

```
(compiler:show-assembly-code)
(compiler:show-assembly-code bool)
```

This parameter controls if the object files produced by the ***STklos*** compiler code must embed a readable version of the code. The code is placed at the beginning of the produced file. This parameter defaults to `#f`.

STklos procedure

```
(compiler:inline-common-functions)
(compiler:inline-common-functions bool)
```

This parameter controls if the compiler must try to inline the most common Scheme primitives (simple arithmetic, main list or vector functions, ...). Code produced when this parameter is set is more efficient. Note that the compiler can sometimes be misleaded if those functions are redefined, hence the existence of this parameter. `compiler:inline-common-functions` is set by default to `#t`.

```
> (compiler:inline-common-functions #t)
> (disassemble-expr '(begin (car '(1 2 3)) (+ a 1)) #t)

000:  CONSTANT             0
002:  IN-CAR
003:  GLOBAL-REF           1
005:  IN-INCR
006:

Constants:
0: (1 2 3)
1: a
```

```
> (compiler:inline-common-functions #f)
> (disassemble-expr '(begin (car '(1 2 3)) (+ a 1)) #t)

000:  PREPARE-CALL
001:  CONSTANT-PUSH        0
003:  GREF-INVOKE          1 1
006:  PREPARE-CALL
007:  GLOBAL-REF-PUSH      2
009:  ONE-PUSH
010:  GREF-INVOKE          3 2
013:

Constants:
0: (1 2 3)
1: car
2: a
3: +
```

STklos procedure

```
(compiler:keep-formals)
(compiler:keep-formals bool)
```

This parameter controls if the formal parameters of a user procedure is kept at runtime. The formal parameters can be accessed with the primitive <<"procedure-formals">>. Default value for `compiler:keep-formals` is `#f`.

```
> (compiler:keep-formals #f)
> (define (foo a b) ( + a b 1))
> foo
#[closure foo]
> (procedure-formals foo)
#f
```

```
> (compiler:keep-formals #t)
> (define (foo a b) ( + a b 1))
> foo
#[closure foo (a b)]
> (procedure-formals foo)
(a b)
```

STklos procedure

```
(compiler:keep-source)
(compiler:keep-source bool)
```

This parameter controls if the source of a user procedure is kept at runtime. The source of a procedure can be accessed with the primitive <<"procedure-source">>. Default value for `compiler:keep-source` is `#f`.

```
> (compiler:keep-source #t)
> (define fib
     (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
> (pretty-print (procedure-source fib))
(lambda (n)
  (if (< n 2)
    n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

Appendix B: GNU Free Documentation License

B.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document ``free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of ``copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The ``Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

B.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section

your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

B.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

B.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents

B.9. TRANSLATION

B.10. TERMINATION

B.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

B.12. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

B.13. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
    Copyright (C)  YEAR  YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

```
    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.