STklos Reference Manual

(version 0.59)

Erick Gallesio
Université de Nice - Sophia Antipolis
930 route des Colles, BP 145
F-06903 Sophia Antipolis, Cedex
France

This document was produced using the Skribe Programming Language and its ConTEXt engine.
For further information on Skribe, see http://www-sop.inria.fr/mimosa/fp/Skribe/.

Document created on November 3, 2004.

This document provides a complete list of procedures and special forms implemented in version **0.59** of STklos. Since STklos is (nearly) compliant with the language described in the Revised^5 Report on the Algorithmic Language Scheme (aka R^{5}RS) [12], the organization of this manual follows closely the one of this document.

1 Introduction

1.1 Overview of STklos

STklos is the successor of STk [6], a Scheme interpreter which was tightly connected to the Tk graphical toolkit [11]. 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+ [2]. 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, 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. See section 4.12 for more information

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 [10].

1.2.3 Other Notations

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

- "`[]`" Brackets are equivalent to parentheses. They are used for grouping and to build lists. A list opened with a left square bracket must be closed with a right square bracket (see section **4.4**).

- "`:`" a colon at the beginning or the end of a symbol introduces a keyword. Keywords are described in section **4.12**.

- `#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#))
```

1.3 Basic Concepts

See the original R^{5}RS document for more informations on the basic concepts of the Scheme Programming Language.

2 Expressions

This chapter describes the main forms available in STklos. R^{5}RS constructions are given very succinctly here for reference. See the (ref :bib "R5RS") for a complete description.

2.1 Literal expressions

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

R^{5}RS *syntax*

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
```

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

2.2 Procedures

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

STklos *syntax*

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)
```

`(closure? obj)` — STklos *procedure*

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

`(case-lambda <clause> ...)` — STklos *syntax*

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

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

R^{5}RS *syntax*

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.

`(setter proc)` — R^{5}RS *procedure*

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)
```

2.4 Conditionals

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

R^{5}RS *syntax*

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

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

`(cond <clause1> <clause2> ...)` R5RS *syntax*

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

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

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

```
(<test> ⇒ <expression>)
```

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

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

A cond expression is evaluated by evaluating the `<test>` expressions of successive `<clause>`s in order until one of them evaluates to a true value When a `<test>` evaluates to a true value, then the remaining `<expression>`s in its `<clause>` are evaluated in order, and the result(s) of the last `<expression>` in the `<clause>` is(are) returned as the result(s) of the entire cond expression. If the selected `<clause>` contains only the `<test>` and no `<expression>`s, then the value of the `<test>` is returned as the result. If the selected `<clause>` uses the ⇒ alternate form, then the `<expression>` is evaluated. Its value must be a procedure that accepts one argument; this procedure is then called on the value of the `<test>` and the value(s) returned by this procedure is(are) returned by the cond expression.

If all `<test>`s evaluate to false values, and there is no else clause, then the result of the conditional expression is *void*; if there is an else clause, then its `<expression>`s are evaluated, and the value(s) of the last one is(are) returned.

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

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

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

`(case <key> <clause1> <clause2> ...)` R5RS *syntax*

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

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

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

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

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

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

`(and <test1> ...)` — R^{5}RS *syntax*

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

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

`(or <test1> ...)` — R^{5}RS *syntax*

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

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

`(when <test> <expression1> <expression2> ...)` STklos *syntax*

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

`(unless <test> <expression1> <expression2> ...)` STklos *syntax*

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

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>)
```
R^{5}RS *syntax*

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

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

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

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

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

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

The second form of `let`, which is generally called a *named let*, is a variant on the syntax of let which provides a more general looping construct than `do` (@pxref{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))
```

`(let* <bindings> <body>)` — R^{5}RS *syntax*

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

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

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

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

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

`(letrec <bindings> <body>)` — R^{5}RS *syntax*

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

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

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

`(fluid-let <bindings> <body>)` — STklos *syntax*

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

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

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

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

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

2.6 Sequencing

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

R^{5}RS *syntax*

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

```
(define x 0)

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

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

2.7 Iterations

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

R^{5}RS *syntax*

`Do` is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits after evaluating the `<expr>`s.

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

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

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

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

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

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

had been written.

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

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

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

STklos *syntax*

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

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

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

STklos *syntax*

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

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

STklos *syntax*

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

2.8 Delayed Evaluation

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

R^{5}RS *procedure*

The `delay` construct is used together with the procedure `force` to implement *lazy evaluation* or *call by need*. `(delay <expression>)` returns an object called a *promise* which at some point in the future may be asked (by the `force` procedure) to evaluate `<expression>`, and deliver the resulting value. The effect of `<expression>` returning multiple values is unpredictable.

See the description of `force` (@pxref{force}) for a more complete description of `delay`.

```
(promise?  obj)
```

STklos *procedure*

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

2.9 Quasiquotation

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

R^{5}RS *syntax*

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

```
‘(list ,(+ 1 2) 4)  ⇒  (list 3 4)
(let ((name ’a)) ‘(list ,name ’,name))
                    ⇒  (list a (quote a))
‘(a ,(+ 1 2) ,@(map abs ’(4 -5 6)) b)
                    ⇒  (a 3 4 5 6 b)
‘((foo ,(- 10 3)) ,@(cdr ’(c)) . )
                    ⇒  ((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.

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

STklos *syntax*

`define-macro` can be used to define low-level macro (i.e. *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)
```

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

R^{5}RS *syntax*

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

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

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

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

`(syntax-rules <literals> <syntax-rule> ...)` — R^{5}RS *syntax*

`<literals>` is a list of identifiers, and each `<syntax-rule>` should be of the form

```
(pattern template)
```

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

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

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

`(let-syntax <bindings> <body>)` — R^{5}RS *syntax*

`<Bindings>` should have the form

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

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

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

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

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

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

`(letrec-syntax <bindings> <body>)` — R^{5}RS *syntax*

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

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

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

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

`(macro-expand form)` — STklos *procedure*

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

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

3 Program structure

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

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

STklos provides a simple module system which is largely inspired from the one of Tung and Dybvig exposed in [14]. As their modules system, STklos modules are defined to be easily used in an interactive environment.

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

STklos *syntax*

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

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

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

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

The `STklos` module, which is predefined, is a special module which contains all the *global variables* of a 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*.

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

`(current-module)` — STklos *procedure*

Returns the current module.

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

`(find-module name)`
`(find-module name default)` — STklos *procedure*

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

`(module? object)` — STklos *procedure*

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

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

`(export <symbol1> <symbol2> ...)` — STklos *syntax*

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

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

The result of `export` is *void*.

`(import <module1> <module2> ...)` — STklos *syntax*

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

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

which exports it.

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

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

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

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

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

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

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

STklos *syntax*

Changes the value of the current module to the module with the given `name`. The expressions evaluated after `select-module` will take place in module `name` environment. Module `name` must have been created previously by a `define-module`. The result of `select-module` is *void*. `Select-module` is particularly useful when debugging since it allows to place toplevel evaluation in a particular module. The following transcript shows an usage of `select-module`.[3]:

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

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

STklos *procedure*

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

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

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

STklos *procedure*

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

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

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

STklos *procedure*

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

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

STklos *procedure*

Returns the list of modules that `module` imports.

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

STklos *procedure*

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

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

STklos *procedure*

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

```
(all-modules)
```

STklos *procedure*

Returns the list of all the living modules.

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

STklos *syntax*

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

This form is in fact a shortcut. In effect,

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

is equivalent to

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

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 (code "eq?").

`(eqv? obj1 obj2)` — R^{5}RS *procedure*

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

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

- `obj1` and `obj2` are both `#t` or both `#f`.
- `obj1` and `obj2` are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                    ⇒  #t
```

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

- `obj1` and `obj2` are both keywords and

```
(string=? (keyword->string obj1)
          (keyword->string obj2))
                    ⇒  #t
```

- `obj1` and `obj2` are both numbers, are numerically equal (see =), and are either both exact or both inexact.
- `obj1` and `obj2` are both characters and are the same character according to the `char=?` procedure (see **char=?**).
- both `obj1` and `obj2` are the empty list.

- `obj1` and `obj2` are pairs, vectors, or strings that denote the same locations in the store.
- `obj1` and `obj2` are procedures whose location tags are equal.

Note: STklos extends 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 :bar)                ⇒  #f
(eqv? '() '())                  ⇒  #t
(eqv? 100000000 100000000)      ⇒  #t
(eqv? (cons 1 2) (cons 1 2))    ⇒  #f
(eqv? (lambda () 1)
      (lambda () 2))            ⇒  #f
(eqv? #f 'nil)                  ⇒  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                   ⇒  #t
```

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

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

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

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

`(eq? obj1 obj2)` — R^{5}RS *procedure*

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

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

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

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

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

`(equal? obj1 obj2)` — R^{5}RS *procedure*

`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?` may fail to terminate if its arguments are circular data structures.

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

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` (same as +nan.0).

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

R5RS *procedure*

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

If `z` is an inexact complex number, then `(real?  z)` is true if and only if `(zero? (imag-part z))` is true. If `x` is an inexact real number, then `(integer?  x)` is true if and only if `(= x (round x))`.

```
(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
```

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

R5RS *procedure*

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

```
(bignum?  x)
```

STklos *procedure*

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
```

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

R5RS *procedure*

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

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

R5RS *procedure*

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

```
(max x1 x2 ...)
(min x1 x2 ...)
```

R5RS *procedure*

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)             ⇒  4    ; exact
(max 3.9 4)           ⇒  4.0  ; inexact
```

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

```
(+ z1 ...)
(* z1 ...)
```

R5RS *procedure*

These procedures return the sum or product of their arguments.

```
(+ 3 4)               ⇒  7
(+ 3)                 ⇒  3
(+)                   ⇒  0
(* 4)                 ⇒  4
(*)                   ⇒  1
```

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

R5RS *procedure*

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

```
(- 3 4)                 ⇒  -1
(- 3 4 5)               ⇒  -6
(- 3)                   ⇒  -3
(/ 3 4 5)               ⇒  3/20
(/ 3)                   ⇒  1/3
```

`(abs x)` — R5RS *procedure*

`Abs` returns the absolute value of its argument.

```
(abs -7)                ⇒  7
```

`(quotient n1 n2)`
`(remainder n1 n2)`
`(modulo n1 n2)` — R5RS *procedure*

These procedures implement number-theoretic (integer) division. n2 should be non-zero. All three procedures return integers.

If `n1/n2` is an integer:

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

If n1/n2 is not an integer:

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

where `nq` is `n1/n2` rounded towards zero, 0 < 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
```

```
(gcd n1 ...)
(lcm n1 ...)
```

R^{5}RS *procedure*

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

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

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

R^{5}RS *procedure*

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

```
(numerator (/ 6 4))  ⇒  3
(denominator (/ 6 4))  ⇒  2
(denominator
(exact->inexact (/ 6 4))) ⇒ 2.0
```

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

R^{5}RS *procedure*

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

Rationale: `Round` rounds to even for consistency with the default rounding mode

specified by the IEEE floating point standard.

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

```
(floor -4.3)          ⇒  -5.0
(ceiling -4.3)        ⇒  -4.0
(truncate -4.3)       ⇒  -4.0
(round -4.3)          ⇒  -4.0

(floor 3.5)           ⇒  3.0
(ceiling 3.5)         ⇒  4.0
(truncate 3.5)        ⇒  3.0
(round 3.5)           ⇒  4.0  ; inexact

(round 7/2)           ⇒  4    ; exact
(round 7)             ⇒  7
```

`(rationalize x y)` — R^{5}RS *procedure*

Rationalize returns the simplest rational number differing from `x` by no more than `y`. A rational number `r1` is simpler than another rational number `r2` if `r1` = `p1/q1` and `r2` = `p2/q2` (in lowest terms) and abs(p1) <= abs(p2) and abs(q1) <= abs(q2). Thus `3/5` is simpler than `4/7`. Although not all rationals are comparable in this ordering (consider `2/7` and `3/5`) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler `2/5` lies between `2/7` and `3/5`). Note that `0` = `0/1` is the simplest rational of all.

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

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

R^{5}RS *procedure*

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

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

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

```
(sqrt z)
```
R5RS *procedure*

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

```
(expt z1 z2)
```
R5RS *procedure*

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

Note: 0^z is 1 if `z = 0` and `0` otherwise.

```
(make-rectangular x1 x2)
(make-polar x3 x)
(real-part z)
(imag-part z)
(magnitude z)
(angle z)
```
R5RS *procedure*

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

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

Then

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

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

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

```
(exact->inexact z)
(inexact->exact z)
```
R5RS *procedure*

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

```
(number->string z)
(number->string z radix)
```
R5RS *procedure*

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

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

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

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

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

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

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

```
(string->number string)
(string->number string radix)
```

R^{5}RS *procedure*

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

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

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

STklos *procedure*

These procedures allow the manipulation of integers as bit fields. The integers can be of arbitrary length. `Bit-and`, `bit-or` and `bit-xor` respectively compute the bitwise

and, inclusive and exclusive *or*. `bit-not` eturns the bitwise *not* of `n`. `bit-shift` returns the bitwise *shift* of `n`. The integer `n` is shifted left by `m` bits; If `m` is negative, `n` is shifted right by `-m` bits.

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

`(random-integer n)` STklos *procedure*

Return an integer in the range `[0, ..., 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 **SRFI-27** (*Source of random bits*) definition for more details).

`(random-real)` STklos *procedure*

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

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.

`(not obj)` R^{5}RS *procedure*

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

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

`(boolean? obj)` R^{5}RS *procedure*

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

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

4.4 Pairs and lists

`(pair? obj)` — R5RS *procedure*

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

`(cons obj1 obj2)` — R5RS *procedure*

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

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

`(car pair)` — R5RS *procedure*

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

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

`(cdr pair)` — R5RS *procedure*

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

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

`(set-car! pair obj)` — R5RS *procedure*

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

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

`(set-cdr! pair obj)` — R5RS *procedure*

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

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

R5RS *procedure*

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

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

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

`(null? obj)` — R5RS *procedure*

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

`(pair-mutable? obj)` — STklos *procedure*

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

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

`(list? obj)` — R5RS *procedure*

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

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

`(list obj ...)` — R5RS *procedure*

Returns a newly allocated list of its arguments.

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

`(list* obj ...)` — STklos *procedure*

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

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

`(length list)` — R^{5}RS *procedure*

Returns the length of `list`.

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

`(append list ...)` — R^{5}RS *procedure*

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

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

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

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

`(append!  list ...)` — STklos *procedure*

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

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

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

`(reverse list)` — R5RS *procedure*

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

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

`(reverse! list)` — STklos *procedure*

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

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

`(list-tail list k)` — R5RS *procedure*

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

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

`(last-pair list)` — STklos *procedure*

Returns the last pair of `list`.

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

`(list-ref list k)` — R5RS *procedure*

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

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

`(memq obj list)`
`(memv obj list)`
`(member obj list)` — R5RS *procedure*

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

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

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

R^{5}RS *procedure*

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

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                ⇒  (a 1)
(assq 'b e)                ⇒  (b 2)
(assq 'd e)                ⇒  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                           ⇒  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                           ⇒  ((a))
(assv 5 '((2 3) (5 7) (11 13)))
                           ⇒  (5 7)
```

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

`(copy-tree obj)`

STklos *procedure*

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

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

STklos *procedure*

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

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

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

`(remove pred list)` — STklos *procedure*

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

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

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

`(delete x list [=])`
`(delete!  x list [=])` — STklos *procedure*

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

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

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

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

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

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 "|".

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

`(symbol? obj)` — R5RS *procedure*

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

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

`(symbol->string string)` — R5RS *procedure*

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

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

`(string->symbol string)` — R5RS *procedure*

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

```
(eq? 'mISSISSIppi 'mississippi)     ⇒  #t
(string->symbol "mISSISSIppi")      ⇒  |mISSISSIppi|
(eq? 'bitBlt (string->symbol "bitBlt"))
                                    ⇒  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog))) ⇒  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
                                    ⇒  #t
```

`(string->unterned-symbol string)` — STklos *procedure*

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

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

`(gensym)`
`(gensym prefix)` — STklos *procedure*

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

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

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	bell
bs	010	backspace	ht	011	tab
nl	012	newline	vt	013	
np	014	page	cr	015	return
so	016		si	017	
dle	020		dc1	021	
dc2	022		dc3	023	
dc4	024		nak	025	
syn	026		etb	027	
can	030		em	031	
sub	032		esc	033	escape
fs	034		gs	035	
rs	036		us	037	
sp	040	space	del	177	delete

```
(char?  obj)
```

R^{5}RS *procedure*

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

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

R^{5}RS *procedure*

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

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

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

R^{5}RS *procedure*

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

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

R5RS *procedure*

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

```
(char->integer char)
(integer->char n)
```

R5RS *procedure*

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

```
(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
```

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

R5RS *procedure*

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

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 "\". The valid escape sequences are shown in the following table.

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

For instance, the string

```
"ab040c\nd
e"
```

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

```
(string?  obj)
```

R^{5}RS *procedure*

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

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

R^{5}RS *procedure*

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

```
(string char ...)
```

R^{5}RS *procedure*

Returns a newly allocated string composed of the arguments.

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

R^{5}RS *procedure*

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

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

R^{5}RS *procedure*

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

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

R^{5}RS *procedure*

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

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

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

R5RS *procedure*

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

`(string<? string1 string2)`
`(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)`

R5RS *procedure*

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

`(substring string start end)`

R5RS *procedure*

`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).

`(string-append string ...)`

R5RS *procedure*

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

`(string->list string)`
`(list->string list)`

R5RS *procedure*

`String->list` returns a newly allocated list of the characters that make up the given string. `List->string` returns a newly allocated string formed from the characters in

the list `list`, which must be a list of characters. `String->list` and `list->string` are inverses so far as `equal?` is concerned.

`(string-copy string)` — R5RS *procedure*

Returns a newly allocated copy of the given `string`.

`(string-split str)`
`(string-split str delimiters)` — STklos *procedure*

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

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

`(string-index str1 str2)` — STklos *procedure*

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

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

`(string-find?  str1 str2)` — STklos *procedure*

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

`(string-fill!  string char)` — STklos *procedure*

Stores `char` in every element of the given `string` and returns *void*.

`(string-mutable?  obj)` — STklos *procedure*

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

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

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

Note: The string SRFI is supported by STklos. The function listed below just don't need to load the full SRFI to be used

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

STklos *procedure*

Returns a string in which the upper case letters of string `str` between the `start` and `end` indices have been replaced by their lower case equivalent. If `start` is 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"
```

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

STklos *procedure*

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

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

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

STklos *procedure*

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

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

STklos *procedure*

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

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

STklos *procedure*

This function returns a string. For every character `c` in the selected range of `str`, if `c` is preceded by a cased character, it is downcased; otherwise it is titlecased. If `start` is 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"
```

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

STklos *procedure*

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

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")
```

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

```
(vector?  obj)
```

R^{5}RS *procedure*

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

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

R^{5}RS *procedure*

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

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

R^{5}RS *procedure*

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

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

`(vector-length vector)` — R5RS *procedure*

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

`(vector-ref vector k)` — R5RS *procedure*

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

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

`(vector-set! vector k obj)` — R5RS *procedure*

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

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

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

`(vector->list vector)`
`(list->vector list)` — R5RS *procedure*

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

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

`(vector-fill! vector fill)` — R5RS *procedure*

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

```
(vector-copy v)
```

STklos *procedure*

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

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

STklos *procedure*

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

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

STklos *procedure*

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

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

```
(sort obj predicate)
```

STklos *procedure*

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

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

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 or a structure.

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

STklos *syntax*

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
```

`(make-struct-type name parent slots)` STklos *procedure*

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.

`(struct-type? obj)` STklos *procedure*

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

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

`(struct-type-slots structype)` STklos *procedure*

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)
```

`(struct-type-slots structype)` STklos *procedure*

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

`(struct-type-name structype)` STklos *procedure*

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

`(struct-type-change-writer! structype proc)` STklos *procedure*

Change the default writer associated to structures of type `structype` to 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 wtructure 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}
```

`(make-struct structype expr ...)` STklos *procedure*

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.

`(struct? obj)` STklos *procedure*

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

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

`(struct-type s)` STklos *procedure*

Returns the structure type of the `s` structure

`(struct-ref s slot-name)` STklos *procedure*

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
```

`(struct-set!  s slot-name value)` — STklos *procedure*

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
```

`(struct-is-a?  s structype)` — STklos *procedure*

Return a boolean that indicates if the structure `s` is a 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 returned by `struct-set!` is *void*.

```
(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
```

`(struct-list s)` — STklos *procedure*

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 Control features

`(procedure?  obj)` — R^{5}RS *procedure*

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

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

`(apply proc arg1 ... args)` — R5RS *procedure*

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

```
(append (list arg1 ...) args)
```

as the actual arguments.

```
(apply + (list 3 4))              ⇒  7

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

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

`(map proc list1 list2 ...)` — R5RS *procedure*

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

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

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

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

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

`(for-each proc list1 list2 ...)` — R5RS *procedure*

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls proc for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to

call proc on the elements of the lists in order from the first element(s) to the last, and the value returned by `for-each` is *void*.

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

`(every pred list1 list2 ...)` STklos *procedure*

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

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

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

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

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

`(any pred list1 list2 ...)` STklos *procedure*

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

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

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

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

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

`(force promise)` — R^{5}RS *procedure*

Forces the value of `promise` (see **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
```

Note: See R^{5}RS for details on a posssible way to implement `force` and `delay`.

`(call-with-current-continuation proc)` — R^{5}RS *procedure*
`(call/cc proc)`

Current version of `call-with-current-continuation` is not conform to R^{5}RS. Furthermore, the current implementation can lead to a fatal error in some circumstances.

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

`(values obj ...)` — R^{5}RS *procedure*

Delivers all of its arguments to its continuation. **Note:** R^{5}RS imposes to use multiple values in the context of of a `call-with-values`. In STklos, if `values` is not used with `call-with-values`, only the first value is used (i.e. others values are *ignored*).

`(call-with-values producer consumer)` — R^{5}RS *procedure*

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

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

(call-with-values * -)                            ⇒  -1
```

`(receive <formals> <expression> <body>)` — STklos *syntax*

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

`(dynamic-wind before thunk after)` — R^{5}RS *procedure*

Current version of `dynamic-wind` mimics the R^{5}RS one. In particular, it does not yet interact with `call-with-current-continuation` as required by R^{5}RS.

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

See R^{5}RS for more details ...

`(eval expression environment)` — R^{5}RS *procedure*
`(eval expression)`

Current form of STklos `eval` is not conform to R^{5}RS.

4.11 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 completely virtual (i.e. the behavior of the port is given by the user).

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

with the string ``| '' (that is a pipe bar followed by a space). Specification of a pipe port can occur everywhere a file name is needed.

4.11.1 Ports

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

R5RS *procedure*

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

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

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

STklos *procedure*

behaves as `call-with-input-file` except that the port passed to `proc` is the 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)
```

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

STklos *procedure*

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

```
(call-with-output-string
  (lambda (x) (write 123 x) (display "Hello" x))) ⇒ "123Hello"
```

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

R5RS *procedure*

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

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

STklos *procedure*

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

```
(input-file-port?  obj)
(output-file-port?  obj)
```
STklos *procedure*

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

```
(interactive-port?  port)
```
STklos *procedure*

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

```
(current-input-port obj)
(current-output-port obj)
```
R^{5}RS *procedure*

Returns the current default input or output port.

```
(current-error-port obj)
```
STklos *procedure*

Returns the current default error port.

```
(with-input-from-file string thunk)
(with-output-to-file string thunk)
```
R^{5}RS *procedure*

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

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

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

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

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

(with-error-to-file string thunk) — STklos *procedure*

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

(with-input-from-string string thunk) — STklos *procedure*

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

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

(with-output-to-string thunk) — STklos *procedure*

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

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

(with-input-from-port port thunk)
(with-output-to-port port thunk)
(with-error-to-port port thunk) — STklos *procedure*

`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

(open-input-file filename) — R^{5}RS *procedure*

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

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

(open-input-string str) — STklos *procedure*

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

`(open-output-file filename)` — R5RS *procedure*

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

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

`(open-output-string)` — STklos *procedure*

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

`(open-file filename mode)` — STklos *procedure*

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

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

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

`(get-output-string port)` — STklos *procedure*

Returns a string containing all the text that has been written on the output string `port`.

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

`(close-input-port port)`
`(close-output-port port)` — R5RS *procedure*

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

```
(close-port port)
```
STklos *procedure*

Closes the port associated with `port`.

```
(rewind-file-port port)
```
STklos *procedure*

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

```
(port-current-line)
(port-current-line port)
```
STklos *procedure*

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

```
(port-current-position)
(port-current-position port)
```
STklos *procedure*

Returns the position associated to the given input `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`.

```
(port-file-name port)
```
STklos *procedure*

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

```
(port-idle-register!  port thunk)
(port-idle-unregister!  port thunk)
(port-idle-reset!  port)
```
STklos *procedure*

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

Hereafter is a (not too realistic) example: a message will be displayed repeatedly until a *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))
```

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

STklos *procedure*

Returns `#t` if `port` is closed and `#f` otherwise.

4.11.2 Input

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

R^{5}RS *procedure*

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

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

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

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

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

STklos *procedure*

`read-with-shared-structure` is identical to `read`. It has been added to be compatible with **SRFI-38** (*External representation of shared structures*). STklos always knew how to deal with recursive input data.

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

STklos *procedure*

This procedure permits to define a new user to reader constructor procedure at runtime. It is defined in **SRFI-10** (*Sharp Comma External Form*) 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)
```

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

R^{5}RS *procedure*

Returns the next character available from the input `port`, updating the `port` to point to the following character. If no more characters are available, an end of file object

is returned. `Port` may be omitted, in which case it defaults to the value returned by `current-input-port`.

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

R5RS *procedure*

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

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

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

R5RS *procedure*

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

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

R5RS *procedure*

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

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

R5RS *procedure*

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

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

```
(read-from-string str)
```

STklos *procedure*

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
```

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

STklos *procedure*

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

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

4.11.3 Output

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

R^{5}RS *procedure*

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

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

STklos *procedure*

Writes a written representation of `obj` to the given port. The main difference with the `write` procedure is that `write*` handles data structures with cycles. Circular structure written by this procedure use the ``#n='' and ``#n#'' notations (see **Circular structure**).

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

STklos *procedure*

`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.

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

R^{5}RS *procedure*

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

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

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

R^{5}RS *procedure*

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

```
(write-char char)
(write-char char port)
```

R^{5}RS *procedure*

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

```
(format port str obj ...)
(format str obj)
```

STklos *procedure*

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`.
- `~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~%")
                              ⇒ "1\n2\n3\n"
(format "~a ~? ~a" 'a "~s" '(new) 'test)
                              ⇒ "a new test"
```

Note: 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`.

Note: Since version 0.58, `format` is also compliant with **SRFI-48** (*Intermediate Format Strings*).

```
(flush-output-port)
(flush-output-port port)
```

STklos *procedure*

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

4.11.4 System interface

```
(load filename)
```

R5RS *procedure*

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

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

```
(try-load filename)
```

STklos *procedure*

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

```
(find-path str)
(find-path str path)
(find-path str path suffixes)
```

STklos *procedure*

In its first form, `find-path` returns the path name of the file that should be loaded by the procedure `load` given the name `str`. The string returned depends of the current load path and of the currently accepted suffixes.

The other forms of `find-path` are more general and allow to give a path list (a list of strings representing supposed directories) and a set of suffixes (given as a list of strings too) to try for finding a file. If no file is found, `find-path` returns `#f`.

For instance, on a "classical" Unix box:

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

```
(require string)
(provide string)
(provided?  string)
```

STklos *procedure*

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

4.12 Keywords

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

`(keyword obj)` — STklos *procedure*

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

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

`(make-keyword s)` — STklos *procedure*

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

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

`(keyword->string key)` — STklos *procedure*

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

`(key-get list key)`
`(key-get list key default)` — STklos *procedure*

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

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

`(key-set!  list key value)` — STklos *procedure*

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

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

```
(key-delete list key)
(key-delete!  list key)
```

STklos *procedure*

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

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

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

4.13 Hash Tables

A hash table consists of zero or more entries, each consisting of a key and a value. Given the key for an entry, the hashing function can very quickly locate the entry, and hence the corresponding value. There may be at most one entry in a hash table with a particular key, but many entries may have the same value.

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

```
(make-hash-table)
(make-hash-table comparison)
(make-hash-table comparison hash)
```

STklos *procedure*

`Make-hash-table` admits three different forms. The most general form admit two arguments. The first argument is a comparison function which determines how keys are compared; the second argument is a function which computes a hash code for an object and returns the hash code as a non negative integer. 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**).
- `comparison` defaults to the `eq?` procedure (see **eq?**).

Consequently,

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

is equivalent to

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

An interesting example is

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

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

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

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

```
(hash-table?  obj)
```

STklos *procedure*

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

```
(hash-table-hash obj)
```

STklos *procedure*

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

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

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

```
(hash-table-put!  hash key value)
```

STklos *procedure*

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

```
(hash-table-get hash key)
(hash-table-get hash key default)
```

STklos *procedure*

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

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

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

`(hash-table-remove hash key)` — STklos *procedure*

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

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

`(hash-table-update!  hash key update-fun init-value)` — STklos *procedure*

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

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

`(hash-table-for-each hash proc)` — STklos *procedure*

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

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

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

`(hash-table-map hash proc)` — STklos *procedure*

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

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

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

`(hash-table->list hash)` — STklos *procedure*

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

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

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

`(hash-table-stats hash)`
`(hash-table-stats hash port)` — STklos *procedure*

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

4.14 Processes

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

`(run-process command p1 p2 ...)` — STklos *procedure*

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

- `:input` permits to redirect the standard input file of the process. Redirection can come from a file or from a pipe. To redirect the standard input from a file, the name of this file must be specified after `:input`. Use the special keyword `:pipe` to redirect the standard input from a pipe.
- `:output` permits to redirect the standard output file of the process. Redirection can go to a file or to a pipe. To redirect the standard output to a file, the name of this file must be specified after `:output`. Use the special keyword `:pipe` to redirect the standard output to a pipe.
- `:error` permits to redirect the standard error file of the process. Redirection can go to a file or to a pipe. To redirect the standard error to a file, the name of this file must be specified after `error`. Use the special keyword `:pipe` to redirect the standard error to a pipe.
- `:wait` must be followed by a boolean value. This value specifies if the process must be run asynchronously or not. By default, the process is run asynchronously (i.e. `:wait` is `#f`).
- `:host` must be followed by a string. This string represents the name of the machine on which the command must be executed. This option uses the external command `rsh`. The shell variable `PATH` must be correctly set for accessing it without specifying its 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")
```

`(process? obj)` — STklos *procedure*

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

`(process-alive? proc)` — STklos *procedure*

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

`(process-pid proc)` — STklos *procedure*

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

`(process-input proc)`
`(process-output proc)`
`(process-error proc)` — STklos *procedure*

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

`(process-wait proc)` STklos *procedure*

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

`(process-exit-status proc)` STklos *procedure*

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

`(process-send-signal proc sig)` STklos *procedure*

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

`(process-kill proc)` STklos *procedure*

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

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

`(process-stop proc)`
`(process-continue proc)` STklos *procedure*

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

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

`(process-list)` STklos *procedure*

Returns the list of processes which are currently running (i.e. alive).

`(fork)`
`(fork thunk)` STklos *procedure*

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 in the parent process is a process object representing the child process and the value returned in 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 in the parent process is a process object representing the child process.

4.15 Sockets

STklos defines **sockets**, on systems which support them, as first class objects. Sockets permits processes to communicate even if they are on different machines. Sockets are useful for creating client-server applications.

```
(make-client-socket hostname port-number)
(make-client-socket hostname port_number line-buffered)
```

STklos *procedure*

`make-client-socket` returns a new socket object. This socket establishes a link between the running program and the application listening on port `port-number` of `hostname`. If the optional argument `line-buffered` has a true value, a line buffered policy is used when writing to the client socket (i.e. characters on the socket are tranmitted as soon as a `#\newline` character is encountered). The default value of `line-buffered` is `#t`.

```
(make-server-socket)
(make-server-socket port-number)
```

STklos *procedure*

`make-server-socket` returns a new socket object. If `port-number` is specified, the socket is listening on the specified port; otherwise, the communication port is chosen by the system.

```
(socket-shutdown sock)
(socket-shutdown sock close)
```

STklos *procedure*

`Socket-shutdown` shutdowns the connection associated to `socket`. If the socket is a server socket, `socket-shutdown` is called on all the clientsockets connected to this server. `Close` indicates if the the socket must be closed or not, when the connection is destroyed. Closing the socket forbids further connections on the same port with the `socket-accept` procedure. Omitting a value for `close` implies the closing of socket.

The following example shows a simple server: when there is a new connection on the port number 12345, the server displays the first line sent to it by the client, discards the others and go back waiting for further client connections.

```
(let ((s (make-server-socket 12345)))
   (let loop ()
      (let ((ns (socket-accept s)))
        (format #t "I've read: ~An"
                (read-line (socket-input ns)))
        (socket-shutdown ns #f)
        (loop))))
```

```
(socket-accept socket)
(socket-accept socket line-buffered)
```

STklos *procedure*

`socket-accept` waits for a client connection on the given `socket`. If no client is already waiting for a connection, this procedure blocks its caller; otherwise, the first connection request on the queue of pending connections is connected and `socket-accept` returns a new client socket to serve this request. This procedure must be called on a server socket created with `make-server-socket`. The result of `socket-accept` is undefined. `Line-buffered` indicates if the port should be considered as a line buffered. If `line-buffered` is omitted, it defaults to `#t`.

The following example is a simple server which waits for a connection on the port 12345[4]. Once the connection with the distant program is established, we read a line on the input port associated to the socket and we write the length of this line on its output port.

```
(let* ((server (make-server-socket 13345))
       (client (socket-accept server))
       (l      (read-line (socket-input client))))
  (format (socket-output client)
          "Length is: ~an" (string-length l))
  (socket-shutdown server))
```

Note that shutting down the `server` socket suffices here to close also the connection to `client`.

`(socket? obj)` — STklos *procedure*

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

`(socket-server? obj)` — STklos *procedure*

Returns `#t` if `socket` is a server socket, otherwise returns `#f`.

`(socket-client? obj)` — STklos *procedure*

Returns `#t` if `socket` is a client socket, otherwise returns `#f`.

`(socket-host-name socket)` — STklos *procedure*

Returns a string which contains the name of the distant host attached to `socket`. If `socket` has been created with `make-client-socket` this procedure returns the official name of the distant machine used for connection. If `socket` has been created with `make-server-socket`, this function returns the official name of the client connected to the socket. If no client has used yet `socket`, this function returns `#f`.

`(socket-host-address socket)` — STklos *procedure*

[4] Under Unix, you can simply connect to a listening socket with the `telnet` command. With the given example, this can be achieved by typing the following command in a window shell:

```
$ telnet localhost 12345
```

Returns a string which contains the IP number of the distant host attached to `socket`. If `socket` has been created with `make-client-socket` this procedure returns the IP number of the distant machine used for connection. If `socket` has been created with `make-server-socket`, this function returns the address of the client connected to the socket. If no client has used yet `socket`, this function returns `#f`.

```
(socket-local-address socket)
```
STklos *procedure*

Returns a string which contains the IP number of the local host attached to `socket`.

```
(socket-port-number socket)
```
STklos *procedure*

Returns the integer number of the port used for `socket`.

```
(socket-input socket)
(socket-output socket)
```
STklos *procedure*

Returns the file port associated for reading or writing with the program connected with `socket`. If no connection has already been established, these functions return `#f`.

The following example shows how to make a client socket. Here we create a socket on port 13 of the machine `kaolin.unice.fr`[5]:

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (format #t "Time is: ~A~%" (read-line (socket-input s)))
  (socket-shutdown  s))
```

4.16 System Procedures

4.16.1 File Primitives

```
(temporary-file-name)
```
STklos *procedure*

Generates a unique temporary file name. The value returned by `temporary-file-name` is the newly generated name of `#f` if a unique name cannot be generated.

```
(rename-file string1 string2)
```
STklos *procedure*

Renames the file whose path-name is `string1` to a file whose path-name is `string2`. The result of `rename-file` is *void*.

```
(remove-file string)
```
STklos *procedure*

[5] Port 13 is generally used for testing: making a connection to it permits to know the distant system's idea of the time of day.

Removes the file whose path name is given in `string`. The result of `remove-file` is *void*.

```
(copy-file string1 string2)
```

STklos *procedure*

Copies the file whose path-name is `string1` to a file whose path-name is `string2`. If the file `string2` already exists, its content prior the call to `copy-file` is lost. The result of `copy-file` is *void*.

```
(file-is-directory?  string)
(file-is-regular?  string)
(file-is-readable?  string)
(file-is-writable?  string)
(file-is-executable?  string)
(file-exists?  string)
```

STklos *procedure*

Returns `#t` if the predicate is true for the path name given in `string`; returns `#f` otherwise (or if `string` denotes a file which does not exist).

```
(getcwd)
```

STklos *procedure*

Returns a string containing the current working directory.

```
(chmod str)
(chmod str option1 ...)
```

STklos *procedure*

Change the access mode of the file whose path name is given in `string`. The options must be composed of either an integer or one of the following symbols `read`, `write` or `execute`. Giving no option to `chmod` is equivalent to pass it the integer `0`. If the operation succeeds, `chmod` returns `#t`; otherwise it returns `#f`.

```
(chmod "~/.stklos/stklosrc" 'read 'execute)
(chmod "~/.stklos/stklosrc" #o644)
```

```
(chdir dir)
```

STklos *procedure*

Changes the current directory to the directory given in string `dir`.

```
(expand-file-name path)
```

STklos *procedure*

`Expand-file-name` expands the filename given in `path` to an absolute path.

```
;; Current directory is ~eg/stklos (i.e. /users/eg/stklos)
(expand-file-name "..")            ⇒ "/users/eg"
(expand-file-name "~eg/../eg/bin") ⇒ "/users/eg/bin"
(expand-file-name "~/stklos)"      ⇒ "/users/eg/stk"
```

```
(canonical-file-name path)
```

STklos *procedure*

Expands all symbolic links in `path` and returns its canonicalized absolute path name. The resulting path does not have symbolic links. If `path` doesn't designate a valid path name, `canonical-file-name` returns `#f`.

`(decompose-file-name string)` STklos *procedure*

Returns an "exploded" list of the path name components given in `string`. The first element in the list denotes if the given `string` is an absolute path or a relative one, being "/" or "." respectively. Each component of this list is a string.

```
(decompose-file-name "/a/b/c.stk") ⇒ ("/" "a" "b" "c.stk")
(decompose-file-name "a/b/c.stk")  ⇒ ("." "a" "b" "c.stk")
```

`(winify-file-name fn)` STklos *procedure*

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 "/Z/temp")
    ⇒ "Z:\\temp"
(list (getcwd) (winify-file-name))
    ⇒ ("//saxo/homes/eg/Projects/,(stklos)"
        "\\\\saxo\\homes\\eg\\Projects\\,(stklos)")
```

`(basename str)` STklos *procedure*

Returns a string containing the last component of the path name given in `str`.

```
(basename "/a/b/c.stk") ⇒ "c.stk"
```

`(dirname str)` STklos *procedure*

Returns a string containing all but the last component of the path name given in `str`.

```
(dirname "/a/b/c.stk") ⇒ "/a/b"
```

`(file-separator)` STklos *procedure*

Retuns the operating system file separator as a character. This is typically `#\/` on Unix system and `#\\` on Windows.

`(make-path dirname name)` STklos *procedure*

Builds a file name from the directory `dirname` and `name`.

`(glob pattern ...)` STklos *procedure*

`Glob` performs file name "globbing" in a fashion similar to the csh shell. `Glob` returns a list of the filenames that match at least one of `pattern` arguments. The `pattern` arguments may contain the following special characters:

- `?` Matches any single character.
- `*` Matches any sequence of zero or more characters.
- `[chars]` Matches any single character in `chars`. If chars contains a sequence of the form `a-b` then any character between `a` and `b` (inclusive) will match.
- \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. First, it does not sort its result list (use the `sort` procedure if you want the list sorted). Second, `glob` only returns the names of files that actually exist; in csh no check for existence is made unless a pattern contains a `?`, `*`, or `[]` construct.

4.16.2 Environment

`(getenv str)`
`(getenv)`

STklos *procedure*

Looks for the environment variable named `str` and returns its value as a string, if it exists. Otherwise, `getenv` returns `#f`. If `getenv` is called without parameter, it returns the list of all the environment variables accessible from the program as an A-list.

```
(getenv "SHELL")
     ⇒ "/bin/zsh"
(getenv)
     ⇒ (("TERM" . "xterm") ("PATH" . "/bin:/usr/bin") ...)
```

`(setenv! var value)`

STklos *procedure*

Sets the environment variable `var` to `value`. `Var` and `value` must be strings. The result of `setenv!` is *void*.

4.16.3 System Informations

`(running-os)` STklos *procedure*

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` or `windows`.

`(hostname)` STklos *procedure*

Return the host name of the current processor as a string.

`(argc)` STklos *procedure*

Returns the number of argument present on the command line

`(argv)` STklos *procedure*

Returns a list of the arguments given on the shell command line. The interpreter options are no included in the result

`(program-name)` STklos *procedure*

Returns the invocation name of the current program as a string.

`(version)` STklos *procedure*

Returns a string identifying the current version of the system. A version is constituted of three numbers separated by a point: the version, the release and sub-release numbers.

`(machine-type)` STklos *procedure*

Returns a string identifying the kind of machine which is running the program. The result string is of the form `‘(os-name)-‘(os-version)-‘(processor-type)`.

`(clock)` STklos *procedure*

Returns an approximation of processor time, in milliseconds, used so far by the program.

`(date)` STklos *procedure*

Returns the current date in a string

`(current-time)` STklos *procedure*

Returns the time since the Epoch (that is 00:00:00 UTC, January 1, 1970), measured in seconds.

`(full-current-time)` STklos *procedure*

Returns the time of the day as a pair where

- the first element is the time since the Epoch (that is 00:00:00 UTC, January 1, 1970), measured in seconds.
- the second element is the number of microseconds in the given second.

`(sleep n)` STklos *procedure*

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.

`(seconds->date sec)` STklos *procedure*

Returns a keyword list for the date given by `sec` (a date based on the Epoch). The keyed values returned are

- second : 0 to 59 (but can be up to 61 to allow for leap seconds)
- minute : 0 to 59
- hour : 0 to 23
- day : 1 to 31
- month : 1 to 12
- year : e.g., 2002
- week-day : 0 (Sunday) to 6 (Saturday)
- year-day : 0 to 365 (365 in leap years)
- dst? : `#t` (daylight savings time) or `#f`
- time-zone-offset : the difference between Coordinated Universal Time (UTC) and local standard time in seconds.

```
(seconds->date (current-time))
            ⇒ (:second 49 :minute 32 :hour 23
               :day 2 :month 4 :year 2002
               :week-day 2 :year-day 91
               :dst #t :time-zone-offset -3600)
```

`(time expr1 expr2 ...)` STklos *syntax*

Evaluates the expressions `expr1`, `expr2`, ... and returns the result of the last expression. This form prints also the time spent for this evaluation on the current error port.

```
(getpid)
```

STklos *procedure*

Returns the system process number of the current program (i.e. the Unix *pid*) as an integer.

4.16.4 Program Arguments Parsing

STklos provides a simple way to parse program arguments with the |parse-arguments| special form. This form is generally used into the |main| function in a Scheme script. See **SRFI-22** (*Running Scheme Scripts on Unix*) on how to use a |main| function in a Scheme program.

```
(parse-arguments <args> <clause1> <clause2> ...)
```

STklos *procedure*

The `parse-arguments` special form is used to parse the command line arguments of a Scheme script. The implementation of this form internally uses the GNU C `getopt` function. As a consequence `parse-arguments` accepts options which start with the '-' (short option) or '–' characters (long option).

The first argument of `parse-arguments` is a list of the arguments given to the program (comprising the program name in the CAR of this list). Following arguments are clauses. Clauses are described later.

By default, `parse-arguments` permutes the contents of (a copy) of the arguments as it scans, so that eventually all the non-options are at the end. However, if the shell environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered.

A clause must follow the syntax:

```
<clause>       ⇒ string | <list-clause>
<list clause>  ⇒ (<option descr> <expr> ...) | (else <expr> ...)
<option descr> ⇒ (<option name> `(<keyword> value)*)
<option name>  ⇒ string
<keyword>      ⇒ :alternate | :arg | :help
```

A string clause is used to build the help associated to the command. A list clause must follow the syntax describes an option. The `<expr>`s associated to a list clauses are executed when the option is recognized. The `else` clauses is executed when all parameters have been parsed. The `:alternate` key permits to have an alternate name for an option (generally a short or long name if the option name is a short or long name). The `:help` is used to provide help about the the option. The `:arg` is used when the option admit a parameter: the symbol given after `:arg` will be bound to the value of the option argument when the corresponding `<expr>`s will be executed.

In an `else` clause the symbol `other-arguments` is bound to the list of the arguments which are not options.

The following example shows a rather complete usage of the `parse-arguments` form

```
#!/usr/bin/env stklos-script

(define (main args)
  (parse-arguments args
     "Usage: foo [options] [parameter ...]"
     "General options:"
       (("verbose" :alternate "v" :help "be more verbose")
         (format #t "Seen the verbose option~%"))
       (("long" :help "a long option alone")
         (format #t "Seen the long option~%"))
       (("s" :help "a short option alone")
         (format #t "Seen the short option~%"))
     "File options:"
       (("input" :alternate "f" :arg file
                 :help "use <file> as input")
         (format #t "Seen the input option with ~S argument~%" file))
       (("output" :alternate "o" :arg file
                  :help "use <file> as output")
         (format #t "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
     (format #t
             "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
```

Note:

- Short option can be concatenated. That is,

  ```
  prog -abc
  ```

  is equivalent to the following program call

  ```
  prog -a -b -c
  ```

- Any argument following a '–' argument is no more considered as an option, even if it starts with a '-' or '–'.
- Option with a parameter can be written in several ways. For instance to set the output in the `bar` file for the previous example can be expressed as
  - `--output=bar`
  - `-o bar`
  - `-obar`

```
(arg-usage port)
(arg-usage port as-sexpr)
```

STklos *procedure*

This procedure is only bound inside a `parse-arguments` form. It pretty prints the help associated to the clauses of the `parse-arguments` form on the given port. If the argument `as-sexpr` is passed and is not `#f`, the help strings are printed on `port` as *S-exprs*. This is useful if the help strings need to be manipulated by a program.

4.16.5 Misc.

```
(system string)
```

STklos *procedure*

Sends the given `string` to the system shell `/bin/sh`. The result of `system` is the integer status code the shell returns.

```
(exec str)
(exec-list str)
```

STklos *procedure*

These procedures execute the command given in `str`. The command given in `str` is passed to `/bin/sh`. `Exec` returns a strings which contains all the characters that the command `str` has printed on it's standard output, whereas `exec-list` returns a list of the lines which constitute the output of `str`.

```
(exec "echo A; echo B")                 ⇒ "A\nB\n"
(exec-list "echo A; echo B")            ⇒ ("A" "B")
```

`(register-exit-function! proc)` — STklos *procedure*

This function registers `proc` as an exit function. This function will be called when the program exits. When called, `proc` will be passed one parmater which is the status given to the `exit` function. The result of `register-exit-function!` is undefined.

```
(let* ((tmp (temporary-file-name))
       (out (open-output-file tmp)))
  (register-exit-function! (lambda (n)
                             (when (zero? n)
                               (remove-file tmp))))
  out)
```

`(exit)`
`(exit ret-code)` — STklos *procedure*

Exits the program with the specified integer return code. If `ret-code` is omitted, the program terminates with a return code of 0. If program has registerd exit functions with `register-exit-function!`, they are called (in an order which is the reverse of their call order).

`(die message)`
`(die message status)` — STklos *procedure*

`Die` prints the given `message` on the current error port and exits the program with the `status` value. If `status` is omitted, it defaults to 1.

`(address-of obj)` — R^{5}RS *procedure*

Returns the address of the object `obj` as an integer.

`(gc)` — R^{5}RS *procedure*

Returns the address of the object `obj` as an integer.

`(void)`
`(void arg1 ...)` — STklos *procedure*

Returns the special *void* object. If arguments are passed to `void`, they are evalued and simply ignored.

```
(error str obj ...)
(error name str obj ...)
```

STklos *procedure*

`error` is used to signal an error to the user. The second form of `error` takes a symbol as first parameter; it is generally used for the name of the procedure which raises the error.

Note: The specification string may follow the *tilde conventions* of `format` (see **format**); in this case this procedure builds an error message according to the specification given in `str`. Otherwise, this procedure is conform to the `error` procedure defined in **SRFI-23** (*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
```

```
(apropos obj)
(apropos obj module)
```

STklos *procedure*

`Apropos` returns a list of symbols whose print name contains the characters of `obj` as a substring . The given `obj` can be a string or symbol. This function returns the list of matched symbols which can be accessed from the given `module` (defaults to the current module if not provided).

```
(trace f-name ...)
```

STklos *syntax*

Invoking `trace` with one or more function names causes the functions named to be traced. Henceforth, whenever such a function is invoked, information about the call and the returned values, if any, will be printed on the current error port.

Calling `trace` with no argument returns the list of traced functions.

`(untrace f-name ...)` — STklos *syntax*

Invoking `untrace` with one or more function names causes the functions named not to be traced anymore.

Calling `untrace` with no argument will untrace all the functions currently traced.

`(pretty-print sexpr :key port width)`
`(pp sexpr :key port width)` — STklos *procedure*

This function tries to obtain a pretty-printed representation of `sexpr`. The pretty-printed form is written on `port` with lines which are no more long than `width` characters. If `port` is omitted 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`.

`(uri-parse str)` — STklos *procedure*

Parses the string `str` as a RFC-2396 URI and return a keyed list with the following components @itemize @item `scheme` : the scheme used as a string (defaults to `"file"`) @item `host` : the host as a string (defaults to "") @item `port` : the port as an integer (0 if no port specified) @item `path` : the path @item `query` : the qury part of the URI as a string (defaults to the empty string) @item `fragment` : the fragment of the URI as a string (defaults to the empty string) @end itemize

```
(uri-parse "http://google.com")
    ⇒ (:scheme "http" :host "google.com" :port 80 :path "/"
        :query "" :fragment "")
(uri-parse "http://stklos.net:8080/a/file?x=1;y=2#end")
    ⇒ (:scheme "http" :host "stklos.net" :port 8080
        :path "/a/file" :query "x=1;y=2" :fragment "end")
(uri-parse "/a/file")
   ⇒ (:scheme "file" :host "" :port 0 :path "/a/file"
       :query "" :fragment "")
(uri-parse "")
   ⇒ (:scheme "file" :host "" :port 0 :path ""
       :query "" :fragment "")
```

`(string->html str)` — STklos *procedure*

This primitive is a convenience function; it returns a string where the HTML special chars are properly translated. It can easily written in Scheme, but this version is fast.

```
(string->html "Just a <test>")
   ⇒ "Just a &lt;test&gt;"
```

4.17 Signals

4.18 Parameter Objects

STklos parameters correspond to the ones defined in **SRFI-39** (*Parameters objects*). See SRFI document for more information.

```
(make-parameter init)
(make-parameter init converter)
```

STklos *procedure*

Returns a new parameter object which is bound in the global dynamic environment to a cell containing the value returned by the call `(converter init)`. If the conversion procedure `converter` is not specified the identity function is used instead.

The parameter object is a procedure which accepts zero or one argument. When it is called with no argument, the content of the cell bound to this parameter object in the current dynamic environment is returned. When it is called with one argument, the content of the cell bound to this parameter object in the current dynamic environment is set to the result of the call `(converter arg)`, where `arg` is the argument passed to the parameter object, and an unspecified value is returned.

```
(define radix
    (make-parameter 10))

(define write-shared
   (make-parameter
      #f
      (lambda (x)
        (if (boolean? x)
            x
            (error '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)       ⇒  ">"
```

```
(parameterize ((expr1 expr2) ...)  <body>)
```

STklos *syntax*

The expressions `expr1` and `expr2` are evaluated in an unspecified order. The value of the `expr1` expressions must be parameter objects. For each `expr1` expression and in an unspecified order, the local dynamic environment is extended with a binding of the parameter object `expr1` to a new cell whose content is the result of the call `(converter val)`, where `val` is the value of `expr2` and converter is the conversion procedure of the parameter object. The resulting dynamic environment is then used for the evaluation of `<body>` (which refers to the R^5RS grammar nonterminal of that name). The result(s) of the parameterize form are the result(s) of the `<body>`.

```
(radix)                                                  ⇒  2
(parameterize ((radix 16)) (radix))                      ⇒  16
(radix)                                                  ⇒  2

(define (f n) (number->string n (radix)))

(f 10)                                                   ⇒  "1010"
(parameterize ((radix 8)) (f 10))                        ⇒  "12"
(parameterize ((radix 8) (prompt (f 10))) (prompt))      ⇒  "1010"
```

`(parameter? obj)` — STklos *procedure*

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

5 Regular Expressions

STklos uses the Philip Hazel's Perl-compatible Regular Expression (PCRE) library for implementing regexps [13]. 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[6]

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:

[6] The latest release of PCRE is available from **http://www.pcre.org/**

\	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 \cx 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 "\\".

Another 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.

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

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

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 \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:

1. 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.

2. 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))+
```

matches "aba" the value of the second captured substring is "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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

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 **5.16** 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 (R), 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 (R), 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 Scheme functions that can use PCRE regexpr described before

```
(string->regexp string)
```

STklos *procedure*

`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.

```
(regexp?  obj)
```

STklos *procedure*

`Regexp` returns `#t` if `obj` is a regexp value created by the `regexp`, otherwise `regexp` returns `#f`.

```
(regexp-match pattern str)
(regexp-match-positions pattern str)
```

STklos *procedure*

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))
```

```
(regexp-replace pattern string substitution)
(regexp-replace-all pattern string substitution)
```

STklos *procedure*

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

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

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

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

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

```
(regexp-quote str)
```

STklos *procedure*

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

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

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

6 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[7].

The technique used is described in details in [4], 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 STklos Pattern Matching Facilities

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

```
(match-case <key> <clause> ...)
```

STklos *syntax*

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

```
(<pattern> <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.

[7] The "pattern matching compiler" has been written by Jean-Marie Geffroy and is part of the Manuel Serrano's Bigloo compiler [1] since several years. The code (and documentation) included in STklos has been stolen from the Bigloo package v2.4 (the only difference between both package is the pattern matching of structures which is absent in STklos).

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)
```

`(match-lambda <clause> ...)` STklos *syntax*

`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
```

6.2 The Pattern Language

The syntax for `<pattern>` is:

```
<pattern> ==>                Matches:

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

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

Explanations Through Examples

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

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

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.

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.

```
(with-handler <handler> <expr1> ...  <exprn>)
```

STklos *syntax*

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"
```

```
(with-exception-handler <handler> <thunk>)
```

STklos *syntax*

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>))
```

(raise obj) — STklos *procedure*

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"
```

(guard (<var> <clause1 > <clause2 > ...) <body>) — STklos *syntax*

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"
```

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.

```
(make-condition-type id parent slot-names)
```

STklos *procedure*

`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.

```
(condition-type?  obj)
```

STklos *procedure*

Returns `#t` if `obj` is a condition type, and `#f` otherwise

```
(make-compound-condition-type id ct1 ...)
```

STklos *procedure*

`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` ...

Note: This function is not defined in **SRFI-34**.

`(make-condition type slot-name value ...)` STklos *procedure*

`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))
```

`(condition?  obj)` STklos *procedure*

Returns `#t` if `obj` is a condition, and `#f` otherwise

`(condition-has-type?  condition condition-type)` STklos *procedure*

`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)
```

`(condition-ref condition slot-name)` STklos *procedure*

`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
```

`(make-compound-condition condition0 condition1 ...)` STklos *procedure*

`Make-compound-condition` returns a compound condition belonging to all condition types that the `conditioni` 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.

`(extract-condition condition condition-type)` STklos *procedure*

`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 |"full-conditions.stk"| is loaded. This can be done with the following call

```
(require "full-conditions")
```

The following hierarchy of conditions is predefined:

```
&condition
   &message (has "message" slot)
   &serious
   &error
      &error-message (has "message" and "location" 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)
```

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 Meta Object Protocol (MOP) [8], in the spirit of the one defined for CLOS [9].

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 [7]. 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 [9]. Some features of STklos are also issued from Dylan [3] or SOS [5].

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 [9] or Dylan [3]. 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

8.2.1.1 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 (superclass1 superclass2 ...)
  (slot-description1
   slot-description2
   ...)
  metaclass option)
```

The *metaclass option* will not be discussed here. The *superclasses* list specifies the super classes of *class* (see **inheritance** 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 **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).

8.2.1.2 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 informations on the standard output:

```
#[<circle> 81aa1f8] is an an instance of class <circle>.
Slots are:
     radius = #[unbound]
     x = 10
     y = 3
```

8.2.1.3 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, the `x` and `y` slots are set to 0 by default. Value of a slot can also be specified by calling `make` with the `:x` and `:y` keywords. 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)        ⇒ 2

(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 coordianate of the `p` point can be done with:

```
(p c)
```

and setting it to 100 can be done using the extended `set!`

```
(set! (x-of p) 100)
```

Note: STklos also define `slot-set!` as the setter function of `slot-ref` (see **setter**). As a consequence, we have:

```
(set! (slot-ref p 'y) 100)
(slot-ref p 'y)       ⇒ 100
```

8.2.1.4 Virtual Slots

Suppose that we need a 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. 4 * Pi)
(radius c)   ⇒ 2.0
```

Of course, we can also used the fucntion `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

8.2.2.1 Class hierarchy and inheritance of slots

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))
```

`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 ??. 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)
```

Note: The order of slots is not significant.

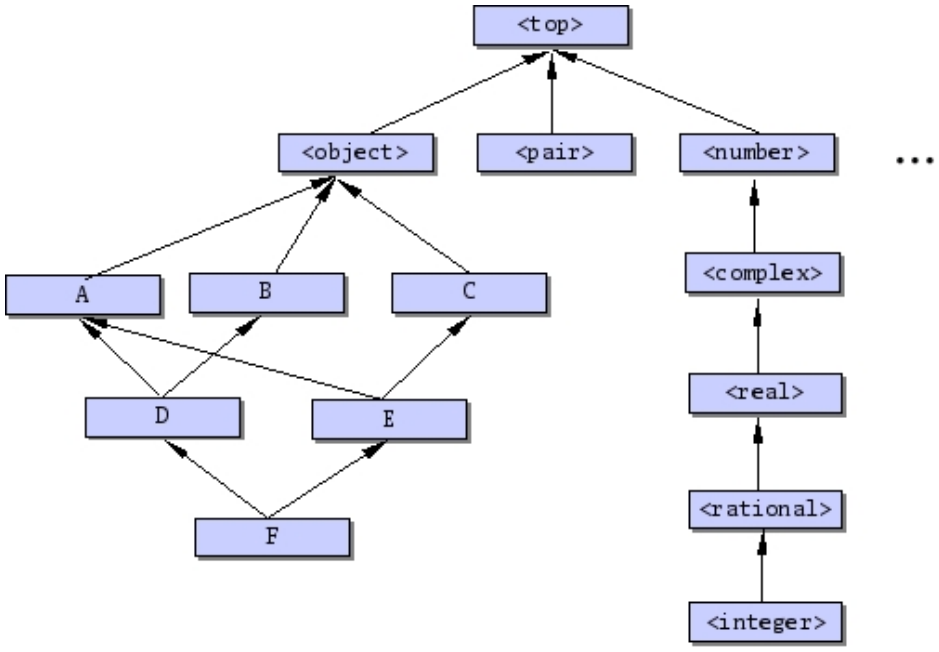

Figure 8.1 a class hierarchy

8.2.2.2 Class precedence list

A class may have more than one superclass.[8]

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.

[8] This section is an adaptation of Jeff Dalton's (J.Dalton@ed.ac.uk) "*Brief introduction to CLOS*" which can be found at the URL **http://www.aiai.ed.ac.uk/~jeff/clos-guide.html**

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 (ref :bib "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 `??`, 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 not too much readable; 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

8.2.3.1 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

1. 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),
2. sorts the list of applicable methods in the "*most specific*" order,
3. 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,

```
(M 1 2)      ⇒ integer-integer
(M 1 1.0)    ⇒ integer-real
(M 'a 1)     ⇒ top-number
(M 1 2 3)    ⇒ three-parameters
```

Notes:

1. 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
```

2. `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.

8.2.3.2 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 **8.2.3.1**). 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)
```

8.2.3.3 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*) to "evaluate" the form when reading it[9].

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))))
```

[9] We suppose here that we are in a context where

```
(define-reader-ctor 'make make)
```

as already been evaluated

8.3 Object System Reference

8.3.1 Class Definition

9 Customizations

STklos environement can be customized using **parameters objects**. These parmaters are listed below.

```
(real-precision)
(real-precision value)
```

STklos *procedure*

This parameter object permits to change the default precision used to print real numbers.

```
(real-precision)         ⇒ 15
(define f 0.123456789)
(display f)              ⊣ 0.123456789
(real-precision 3)
(display f)              ⊣ 0.123
```

```
(load-path)
(load-path value)
```

STklos *procedure*

`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 *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.

```
(load-suffixes)
(load-suffixes value)
```

STklos *procedure*

`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.

```
(load-verbose)
(load-verbose value)
```

STklos *procedure*

`load-verbose` is a parameter object. It permits to display the path name of the files whch 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.

10 Using the SLIB package

Aubrey Jaffer maintains a package called ***SLIB*** which is a portable Scheme library which provides compatibility and utility functions for all standard Scheme implementations. To use this package, you have just to type

```
(require "slib")
```

and follow the instructions given in the *SLIB* library to use a particular package.

Note: *SLIB* uses also the *require/provide* mechanism to load components of the library. Once *SLIB* has been loaded, the standard STklos `require` and `provide` are overloaded such as if their parameter is a string this is the old STklos procedure which is called, and if their parameter is a symbol, this is the *SLIB* one which is called.

11 SRFIs

The *Scheme Request for Implementation* (**SRFI**) process grew out of the Scheme Workshop held in Baltimore, MD, on September 26, 1998, where the attendees considered a number of proposals for standardized feature sets for inclusion in Scheme implementations. Many of the proposals received overwhelming support in a series of straw votes. Along with this there was concern that the next Revised Report would not be produced for several years and this would prevent the timely implementation of standardized approaches to several important problems and needs in the Scheme community.

Only the implemented SRFIs are (briefly) presented here. For further information on each SRFI, please look at the official **SRFI site**.

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*). Furthermore, the feature identifier *stklos* is defined for application which need to know on which Scheme implementation they are running on.

SRFI-1 – List Library

SRFI-1 defines an extensive library for list manipulation. The implementation used in STklos is based on the reference implementation from Olin Shivers. To use, SRFI-1 you need to insert the following expression

```
(require "srfi-1")
```

in your code or uses the `cond-expand` special form.

SRFI-2 – AND-LET*: an AND with local bindings, a guarded LET* special form

SRFI-2 defines an *and* form with local binding which acts as a guarded *let**. To use, SRFI-2 you need to insert the following expression

```
(require "srfi-2")
```

in your code or uses the `cond-expand` special form.

SRFI-4 – Homogeneous numeric vector datatypes

SRFI-4 defines a set of data types for vectors whose element are of the same numeric type (homogeneous vectors). To use SRFI-4, you need to insert the following expression

```
(require "srfi-4")
```

in your code or uses the `cond-expand` special form.

SRFI-6 – Basic String Ports

SRFI-6 is fully supported and is completely described in this document (procedures `open-input-string`, `open-output-string` and `get-output-string`).

SRFI-7 – Feature-based program configuration language

SRFI-7 is fully supported. To use SRFI-7, you need to insert the following expression

```
(require "srfi-7")
```

in your code or uses the `cond-expand` special form.

SRFI-8 – receive: Binding to multiple values

SRFI-8 is fully supported and is completely described in this document (special form `receive`)

SRFI-9 – Defining Record Types

SRFI-9 is fully supported (the implementation uses STklos classes to implement SRFI-9 records). To use SRFI-9, you need to insert the following expression

```
(require "srfi-9")
```

in your code or uses the `cond-expand` special form.

SRFI-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 **define-reader-ctor**).

SRFI-11 – Syntax for receiving multiple values

SRFI-11 is fully supported. To use SRFI-11, you need to insert the following expression

```
(require "srfi-11")
```

in your code or uses the `cond-expand` special form.

SRFI-13 – String Library

SRFI-13 is fully supported. To use SRFI-13, you need to insert the following expression

```
(require "srfi-13")
```

in your code or uses the `cond-expand` special form.

SRFI-14 – Character-Set Library

SRFI-14 is fully supported. To use SRFI-14, you need to insert the following expression

```
(require "srfi-14")
```

in your code or uses the `cond-expand` special form.

SRFI-16 – Syntax for procedures of variable arity

SRFI-16 is fully supported and is completely described in this document (procedure `case-lambda`).

SRFI-17 – Generalized set!

SRFI-17 is fully supported and is completely described in this document (procedures `set!` and `setter`). However, the following expression

```
(require "srfi-17")
```

in your code (or the use of the `cond-expand` special form) permits to define the setters for the (numerous) cXXXXr list procedures.

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 recommends to invoke the Scheme script interpreter from the script via a /usr/bin/env trampoline, like this:

```
#!/usr/bin/env <executable>
```

where `<executable>` can recover several specified names. STklos uses only the name `stklos-script` for `<executable>`.

Here is an example of the classical `echo` command (without option) in Scheme:

```
#!/usr/bin/env stklos-script

(define (main arguments)
  (for-each (lambda (x) (display x) (display #space))
            (cdr arguments))
  (newline)
  0)
```

SRFI-23 – Error reporting mechanism

SRFI-23 is fully supported. See the documentation of the (ref :mark "error") primitive form more information (in fact STklos `error` is more general than the one defined in SRFI-23).

SRFI-26 – Notation for Specializing Parameters without Currying

SRFI-26 is fully supported. To use SRFI-31, you need to insert the following expression

```
(require "srfi-26")
```

in your code or uses the `cond-expand` special form.

SRFI-27 – Source of random bits

SRFI-27 is fully supported. See **random-integer** and **random-real**.

SRFI-28 – Basic Format Strings

SRFI-28 is fully supported. See the documentation of the **format** primitive form more information (in fact STklos `format` is more general than the one defined in **SRFI-28**).

SRFI-30 – Nested Multi-line Comments

SRFI-30 is fully supported by STklos reader.

SRFI-31 – A special form for recursive evaluation

SRFI-31 is fully supported. To use SRFI-31, you need to insert the following expression

```
(require "srfi-31")
```

in your code or uses the `cond-expand` special form.

SRFI-34 – Exception Handling for Programs

SRFI-34 is fully supported and is completely described in this document (see `with-exception-handler` and `guard`).

SRFI-35 – Conditions

SRFI-35 is fully supported. To use SRFI-35, you need to insert the following expression

```
(require "srfi-35")
```

in your code or uses the `cond-expand` special form. 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. To use SRFI-36, you need to insert the following expression

```
(require "srfi-36")
```

in your code or uses the `cond-expand` special form. See section **7.3** for the predefined conditions and when it is required to load this file.

SRFI-38 – External representation of shared structures

SRFI-38 is fully supported by STklos reader.

SRFI-39 – Parameters objects

SRFI-39 is fully supported and is completely described in this document (procedures **make-parameter** and **parameterize**).

SRFI-48 – Intermediate Format Strings

SRFI-48 is fully supported and is completely described in this document (procedure `format`).

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Bibliography

[1] – **Bigloo Home Page**.

[2] – **The GTK+ Toolkit Home Page**.

[3] Apple Computer – **Dylan: an Object Oriented Dynamic Language** – *Apple*, April, 1992.

[4] 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.

[5] Chris Hanson – **The SOS Reference Manual, version 1.5** – March, 1993.

[6] Erick Gallesio – **Reference Manual** – RT 95-31a, I3S CNRS / Universit351 de Nice - Sophia Antipolis, juillet, 1995, pp. 82.

[7] Gregor Kickzales – **Tiny-Clos** – December, 1992.

[8] Gregor Kickzales, Jim de Rivières and Daniel G. Bobrow – **The Art of Meta Object Protocol** – *MIT Press*, 1991.

[9] Guy L. Steele Jr. – **Common Lisp: the Language, 2nd Edition** – *Digital Press*, 12 Crosby Drive, Bedford, MA 01730, USA, 1990.

[10] ISO/IEC – **Information technology, Processing Languages, Document Style Semantics and Specification Languages (DSSSL)** – 10179:1996(E), ISO, , 1996.

[11] John K. Ousterhout – **An X11 toolkit based on the Tcl Language** – USENIX Winter Conference, January, 1991, pp. 105–115.

[12] 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.

[13] Philip Hazel – **PCRE (Perl Compatible Regular Expressions) Home page**.

[14] Sho-Huan Simon Tung and R. Kent Dybvig – **Reliable Interactive Programming with Modules** – LISP and Symbolic Computation, 91996, pp. 343–358.