STK Reference manual

Version 3.0

Erick Gallesio
Université de Nice - Sophia Antipolis
Laboratoire I3S - CNRS URA 1376 - ESSI.
Route des Colles
B.P. 145
06903 Sophia-Antipolis Cedex - FRANCE

email: eg@unice.fr

January 1996

Document Reference

Erick Gallesio, *STk Reference Manual*, RT 95-31a, I3S-CNRS / Université de Nice - Sophia Antipolis, juillet 1995.

Contents

Part I

Reference Manual

Introduction

This document provides a complete list of procedures and special forms implemented in version 3.0 of STk. Since STk is (nearly) compliant with the language described in the *Revised*^{4} *Report on the Algorithmic Language Scheme* (denoted R^4RS hereafter[1])[1], the organization of this manual follows the R^4RS and only describes extensions.

1 Overview of STk

Today's graphical toolkits for applicative languages are often not satisfactory. Most of the time, they ask the user to be an X window system expert and force him/her to cope with arcane details such as server connections and event queues. This is a real problem, since programmers using this kind of languages are generally not inclined to system programming, and few of them will bridge the gap between the different abstraction levels.

Tk is a powerful graphical toolkit promising to fill that gap. It was developed at the University of Berkeley by John Ousterhout [2]. The toolkit offers high level widgets such as buttons or menus and is easily programmable, requiring little knowledge of X fundamentals. Tk relies on an interpretative shell-like language named Tcl [3].

STk is an implementation of the Scheme programming language, providing a full integration of the Tk toolkit. In this implementation, Scheme establishes the link between the user and the Tk toolkit, replacing Tcl.

2 Lexical conventions

2.1 Identifiers

Syntactic keywords can be used as variables in STk. Users must be aware that this extension of the language could lead to ambiguities in some situations.

2.2 Comments

As in the R^4RS, a semicolon (;) indicates the start of a comment. In STk, comments can also be introduced by `#!`. This extension is particularly useful for building STk scripts. On most Unix implementations, if the first line of a script looks like this:

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

then the script can be started directly as if it were a binary. STk is loaded behind the scenes and reads and executes the script as a Scheme program. Of course this assumes that STk is located in `/usr/local/bin`.

2.3 Other notations

STk accepts all the notations defined in R^4RS plus

`[ ]` Brackets are equivalent to parentheses. They are used for grouping and to notate lists. A list opened with a left square bracket must be closed with a right square bracket (section 6.3).

`:` A colon at the beginning of a symbol introduces a keyword. Keywords are described in section 6.11.

[1] The *Revised*^{4} *Report on the Algorithmic Language Scheme* is available through anonymous FTP from `ftp.cs.indiana.edu` in the directory `/pub/scheme-repository/doc`

`#.<expr>` is read as the evaluation of the Scheme expression `<expr>`. The evaluation is done during the `read` process, when the `#.` is encountered. Evalustaion is done in the global-environment.

```
(define foo 1)
#.foo
      ⟹ 1
'(foo #.foo #.(+ foo foo))
      ⟹ (foo 1 2)
(let ((foo 2))
   #.foo)
      ⟹ 1
```

3 Basic concepts

Identical to R^4RS.

4 Expressions

4.1 Primitive expression types

`(quote` ⟨datum⟩`)` syntax
`'`⟨datum⟩ syntax

The quoting mechanism is identical to R^4RS. Keywords (see section 6.11), as numerical constants, string constants, character constants, and boolean constants evaluate "to themselves"; they need not be quoted.

```
'"abc"          ⟹  "abc"
"abc"           ⟹  "abc"
'145932         ⟹  145932
145932          ⟹  145932
'#t             ⟹  #t
#t              ⟹  #t
':key           ⟹  :key
:key            ⟹  :key
```

Note: R^4RS requires to quote constant lists and constant vectors. This is not necessary with STk.

(⟨operator⟩ ⟨operand$_1$⟩ ...) syntax

Identical to R^4RS. Furthermore, ⟨operator⟩ can be a macro (see section 6.14).

`(lambda` ⟨formals⟩ ⟨body⟩`)` syntax
`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)` syntax
`(if` ⟨test⟩ ⟨consequent⟩`)` syntax
`(set!` ⟨variable⟩ ⟨expression⟩`)` syntax

Identical to R^4RS.

4.2 Derived expression types

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntax
`(case` ⟨key⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntax
`(and` ⟨test$_1$⟩ ...`)` syntax
`(or` ⟨test$_1$⟩ ...`)` syntax

Identical to R^4RS.

`(when` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` 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.

`(unless` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` 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.

`(let` ⟨bindings⟩ ⟨body⟩`)` syntax
`(let` ⟨variable⟩ ⟨bindings⟩ ⟨body⟩`)` syntax
`(let*` ⟨bindings⟩ ⟨body⟩`)` syntax

Identical to R^4RS.

`(fluid-let` ⟨bindings⟩ ⟨body⟩`)` 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 a
        (fluid-let ((a 'in)) (f))
        a))
          ⟹ (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 behaviour

```
(let ((cont #f)
      (l    '())
      (a    'out))

  (set! l (cons a l))
  (fluid-let ((a 'in))
    (set! cont (call/cc (lambda (k) k)))
    (set! l (cons a l)))
  (set! l (cons a l))

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

`(letrec` ⟨bindings⟩ ⟨body⟩`)` syntax
`(begin` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax
`(do` ⟨inits⟩ ⟨test⟩ ⟨body⟩`)` syntax
`(delay` ⟨expression⟩`)` syntax
`(quasiquote` ⟨template⟩`)` syntax
`` ` ⟨template⟩ syntax

Identical to R^4RS.

`(dotimes` *(var count)* ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax
`(dotimes` *(var count result)* ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax

`Dotimes` evaluates the *count* form, which must return an integer. It then evaluates the ⟨expression⟩s once for each integer from zero (inclusive) to *count* (exclusive), in order, with the variable *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` expression. If *result* is omitted, `dotimes` returns `#f`.

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

`(while` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax

`While` evaluates the ⟨expression⟩s until ⟨test⟩ returns a false value. The value of a `while` construct is unspecified.

`(until` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax

`Until` evaluates the ⟨expression⟩s while ⟨test⟩ returns a false value. The value of an `unless` construct is unspecified.

5 Program structure

Identical to R^4RS.

6 Standard procedures

6.1 Booleans

In STk the boolean value `#f` is different from the empty list, as required by R^4RS.

`(not` *obj*`)` procedure
`(boolean?` *obj*`)` procedure

Identical to R^4RS.

6.2 Equivalence predicates

`(eqv?` *obj*$_1$ *obj*$_2$`)` procedure

STk extends the `eqv?` predicate defined in the R^4RS to take keywords into account: if *obj*$_1$ and *obj*$_2$ are both keywords, the `eqv?` predicate will yield `#t` if and only if

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

`(eq?` *obj*$_1$ *obj*$_2$`)` procedure

STk extends the `eq?` predicate defined in R^4RS to take keywords into account. On keywords, `eq?` behaves like `eqv?`.

```
(eq? :key :key)           ⟹  #t
```

`(equal?` *obj*$_1$ *obj*$_2$`)` procedure

Identical to R^4RS.

6.3 Pairs and lists

`(pair?` *obj*`)` procedure
`(cons` *obj*$_1$ *obj*$_2$`)` procedure
`(car` *pair*`)` procedure
`(cdr` *pair*`)` procedure
`(set-car!` *pair obj*`)` procedure
`(set-cdr!` *pair obj*`)` procedure
`(caar` *pair*`)` procedure
`(cadr` *pair*`)` procedure
⋮ ⋮
`(cdddar` *pair*`)` procedure
`(cddddr` *pair*`)` procedure
`(null?` *obj*`)` procedure
`(list?` *obj*`)` procedure
`(list` *obj* ...`)` procedure
`(length` *list*`)` procedure
`(append` *list* ...`)` procedure
`(reverse` *list*`)` procedure
`(list-tail` *list k*`)` procedure
`(list-ref` *list k*`)` procedure
`(memq` *obj list*`)` procedure
`(memv` *obj list*`)` procedure
`(member` *obj list*`)` procedure
`(assq` *obj alist*`)` procedure
`(assv` *obj alist*`)` procedure
`(assoc` *obj alist*`)` procedure

Identical to R^4RS.

`(list*` *obj*`)` 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)
```

`(copy-tree` *obj*`)` 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.

6.4 Symbols

The STk reader can cope with 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 "`|`"

```
'|x|                       ⟹ x
(string->symbol "X")       ⟹ |X|
(symbol->string '|X|)      ⟹ "X"
'|a  b|                    ⟹ |a  b|
'a|B|c                     ⟹ |aBc|
(write '|FoO|)             ⟹ writes the string "|FoO|"
(display '|FoO|)           ⟹ writes the string "FoO"
```

Note: This notation has been introduced because R^4RS states that case must not be significant in symbols whereas the Tk toolkit is case significant (or more precisely thinks it runs over Tcl which is case significant). However, symbols containing the character "`|`" itself still can't be read in.

`(symbol?` *obj*`)` 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` *symbol*`)` procedure
`(string->symbol` *string*`)` procedure

Identical to R^4RS.

`(gensym)` procedure
`(gensym` *prefix*`)` procedure

`Gensym` 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 a string.

```
(gensym)
          ⟹ |G100|
(gensym "foo-")
          ⟹ foo-101
```

6.5 Numbers

The only numbers recognized by STK are integers (with arbitrary precision) and reals (implemented as C `double floats`).

`(number?` *obj*`)` procedure

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

`(complex?` *obj*`)` procedure

Returns the same result as *number?*. Note that complex numbers are not implemented.

`(real?` *obj*`)` procedure

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

`(rational?` *obj*`)` procedure

Returns the same result as *number?*. Note that rational numbers are not implemented.

`(integer?` *obj*`)` procedure

Returns `#t` if *obj* is an integer, otherwise returns `#f`. *Note:* The STK interpreter distinguishes between integers which fit in a C `long int` (minus 8 bits) and integers of arbitrary length (aka "bignums"). This should be transparent to the user, though.

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

In this implementation, integers (C `long int` or "bignums") are exact numbers and floats are inexact.

`(=` z_1 z_2 z_3 ...`)` procedure
`(<` x_1 x_2 x_3 ...`)` procedure
`(>` x_1 x_2 x_3 ...`)` procedure
`(<=` x_1 x_2 x_3 ...`)` procedure
`(>=` x_1 x_2 x_3 ...`)` procedure
`(zero?` z`)` procedure
`(positive?` z`)` procedure
`(negative?` z`)` procedure
`(odd?` z`)` procedure
`(even?` z`)` procedure
`(max` x_1 x_2 ...`)` procedure
`(min` x_1 x_2 ...`)` procedure
`(+` z_1 ...`)` procedure
`(*` z_1 ...`)` procedure
`(-` z_1 z_2`)` procedure
`(-` z`)` procedure
`(-` z_1 z_2 ...`)` procedure

(/ z_1 z_2) procedure
(/ z) procedure
(/ z_1 z_2 ...) procedure
(abs x) procedure
(quotient n_1 n_2) procedure
(remainder n_1 n_2) procedure
(modulo n_1 n_2) procedure
(gcd n_1 ...) procedure
(lcm n_1 ...) procedure

Identical to R^4RS.

(numerator q) procedure
(denominator q) procedure

Not implemented.

(floor x) procedure
(ceiling x) procedure
(truncate x) procedure
(round x) procedure

Identical to R^4RS.

(rationalize x y) procedure

not yet implemented.

(exp z) procedure
(log z) procedure
(sin z) procedure
(cos z) procedure
(tan z) procedure
(asin z) procedure
(acos z) procedure
(atan z) procedure
(atan y x) procedure
(sqrt z) procedure
(expt z_1 z_2) procedure

Identical to R^4RS.

(make-rectangular x_1 x_2) procedure
(make-polar x_1 x_2) procedure
(real-part z) procedure
(imag-part z) procedure
(magnitude z) procedure
(angle z) procedure

These procedures are not implemented since complex numbers are not defined.

(exact->inexact z) procedure
(inexact->exact z) procedure
(number->string *number*) procedure
(number->string *number radix*) procedure
(string->number *string*) procedure
(string->number *string radix*) procedure

Identical to R^4RS.

name	*value*	*alternate name*	*name*	*value*	*alternate name*
nul	000	null	bs	010	backspace
soh	001		ht	011	tab
stx	002		nl	012	newline
etx	003		vt	013	
eot	004		np	014	page
enq	005		cr	015	return
ack	006		so	016	
bel	007	bell	si	017	
dle	020		can	030	
dc1	021		em	031	
dc2	022		sub	032	
dc3	023		esc	033	escape
dc4	024		fs	034	
nak	025		gs	035	
syn	026		rs	036	
etb	027		us	037	
sp	040	space			
del	177	delete			

Table .1: Valid character names

6.6 Characters

Table 1 gives the list of allowed character names together with their ASCII equivalent expressed in octal.

`(char?` *obj*`)` procedure
`(char=?` $char_1$ $char_2$`)` procedure
`(char<?` $char_1$ $char_2$`)` procedure
`(char>?` $char_1$ $char_2$`)` procedure
`(char<=?` $char_1$ $char_2$`)` procedure
`(char>=?` $char_1$ $char_2$`)` procedure
`(char-ci=?` $char_1$ $char_2$`)` procedure
`(char-ci<?` $char_1$ $char_2$`)` procedure
`(char-ci>?` $char_1$ $char_2$`)` procedure
`(char-ci<=?` $char_1$ $char_2$`)` procedure
`(char-ci>=?` $char_1$ $char_2$`)` procedure
`(char-alphabetic?` *char*`)` procedure
`(char-numeric?` *char*`)` procedure
`(char-whitespace?` *char*`)` procedure
`(char-upper-case?` *letter*`)` procedure
`(char-lower-case?` *letter*`)` procedure
`(char->integer` *char*`)` procedure
`(integer->char` *n*`)` procedure
`(char-upcase` *char*`)` procedure
`(char-downcase` *char*`)` procedure

Identical to R^4RS.

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

Table .2: String escape sequences

6.7 Strings

STk string constants allow the insertion of arbitrary characters by encoding them as escape sequences, introduced by a backslash (\). The valid escape sequences are shown in Table 2. For instance, the string

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

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

`(string?` *obj*`)` procedure
`(make-string` *k*`)` procedure
`(make-string` *k char*`)` procedure
`(string` *char ...*`)` procedure
`(string-length` *string*`)` procedure
`(string-ref` *string k*`)` procedure
`(string-set!` *string k char*`)` procedure
`(string=?` $string_1$ $string_2$`)` procedure
`(string-ci=?` $string_1$ $string_2$`)` procedure
`(string<?` $string_1$ $string_2$`)` procedure
`(string>?` $string_1$ $string_2$`)` procedure
`(string<=?` $string_1$ $string_2$`)` procedure
`(string>=?` $string_1$ $string_2$`)` procedure
`(string-ci<?` $string_1$ $string_2$`)` procedure
`(string-ci>?` $string_1$ $string_2$`)` procedure
`(string-ci<=?` $string_1$ $string_2$`)` procedure
`(string-ci>=?` $string_1$ $string_2$`)` procedure
`(substring` *string start end*`)` procedure
`(string-append` *string ...*`)` procedure
`(string->list` *string*`)` procedure
`(list->string` *chars*`)` procedure
`(string-copy` *string*`)` procedure
`(string-fill!` *string char*`)` procedure

Identical to R^4RS.

`(string-find?` $string_1$ $string_2$`)` procedure

Returns `#t` if $string_1$ appears somewhere in $string_2$; otherwise returns `#f`.

`(string-index` *string*$_1$ *string*$_2$`)` procedure

Returns the index of where *string*$_1$ is a substring of *string*$_2$ if it exists; returns `#f` otherwise.

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

`(string-lower` *string*`)` procedure

Returns a string in which all upper case letters of `string` have been replaced by their lower case equivalent.

`(string-upper` *string*`)` procedure

Returns a string in which all lower case letters of `string` have been replaced by their upper case equivalent.

6.8 Vectors

`(vector?` *obj*`)` procedure
`(make-vector` *k*`)` procedure
`(make-vector` *k fill*`)` procedure
`(vector` *obj ...*`)` procedure
`(vector-length` *vector*`)` procedure
`(vector-ref` *vector k*`)` procedure
`(vector-set!` *vector k obj*`)` procedure
`(vector->list` *vector*`)` procedure
`(list->vector` *list*`)` procedure
`(vector-fill!` *vector fill*`)` procedure

Identical to R^{4}RS.

`(vector-copy` *vector*`)` procedure

returns a copy of *vector*.

`(vector-resize` *vector size*`)` procedure

vector-resize physically changes the size of *vector*. If *size* is greater than the old vector size, the contents of the newly allocated cells are undefined.

6.9 Control features

`(procedure?` *obj*`)` procedure
`(apply` *proc args*`)` procedure
`(apply` *proc arg*$_1$ *... args*`)` procedure
`(map` *proc list*$_1$ *list*$_2$ *...*`)` procedure
`(for-each` *proc list*$_1$ *list*$_2$ *...*`)` procedure
`(force` *promise*`)` procedure

Identical to R^{4}RS.

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

`Call/cc` is a shorter name for `call-with-current-continuation`.

`(closure?` *obj*`)` procedure

returns `#t` if *obj* is a procedure created by evaluating a lambda expression, otherwise returns `#f`.

`(primitive?` *obj*`)` procedure

returns `#t` if *obj* is a procedure and is not a closure, otherwise returns `#f`.

`(promise?` *obj*`)` procedure

returns `#t` if *obj* is an object returned by the application of `delay`, otherwise returns `#f`.

`(continuation?` *obj*`)` procedure

returns `#t` if *obj* is a continuation obtained by `call/cc`, otherwise returns `#f`.

`(dynamic-wind` ⟨thunk$_1$⟩ ⟨thunk$_2$⟩ ⟨thunk$_3$⟩`)` procedure

⟨Thunk$_1$⟩, ⟨thunk$_2$⟩ and ⟨thunk$_3$⟩ are called in order. The result of `dynamic-wind` is the value returned by ⟨thunk$_2$⟩. If ⟨thunk$_2$⟩ escapes from its continuation during evaluation (by calling a continuation obtained by `call/cc` or on error), ⟨thunk$_3$⟩ is called. If ⟨thunk$_2$⟩ is later reentered, ⟨thunk$_1$⟩ is called.

`(catch` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` syntax

The ⟨expression⟩s are evaluated from left to right. If an error occurs, evaluation of the ⟨expression⟩s is aborted, and `#t` is returned to `catch`'s caller. If evaluation finishes without an error, `catch` returns `#f`.

```
(let* ((x 0)
       (y (catch
            (set! x 1)
            (/ 0) ; causes a "division by 0" error
            (set! x 2))))
  (cons x y))
        ⟹ (1 . #t)
```

`(procedure-body` ⟨procedure⟩`)` procedure

returns the body of ⟨procedure⟩. If ⟨procedure⟩ is not a closure, `procedure-body` returns `#f`.

```
(define (f a b)
   (+ a (* b 2)))

(procedure-body f)          ⟹ (lambda (a b)
                                 (+ a (* b 2)))
(procedure-body car)        ⟹ #f
```

6.10 Input and output

The *R4RS* states that ports represent input and output devices. However, it defines only ports which are attached to files. In STk, ports can also be attached to strings or to a external command input or output. String ports are similar to file ports, except that characters are read from (or written to) a string rather than a file. External command input or output ports are implemented with Unix pipes and are called pipe ports. A pipe port is created by specifying the command to execute prefixed with the string `"| "`. Specification of a pipe port can occur everywhere a file name is needed.

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

Note: if *s*tring starts with the two characters `"| "`, these procedures return a pipe port. Consequently, it is not possible to open a file whose name starts with those two characters.

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

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

```
(call-with-input-string "123 456" (lambda (x) (read x)))
          ⟹ 123
```

`(call-with-output-string` *proc*`)` 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*`)` procedure
`(output-port?` *obj*`)` procedure

Identical to R4RS.

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

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

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

Identical to R4RS.

`(current-error-port)` procedure

Returns the current default error port.

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

Identical to R4RS.

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 ouput 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()
    (format #t "A simple mail sent from STk\n")))
```

`(with-input-from-string` *string thunk*`)` 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 yielded by *thunk*.

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

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

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

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

`(open-input-file` *filename*`)` procedure
`(open-output-file` *filename*`)` procedure

Identical to R^4RS.
Note: if *f*ilename starts with the string `"| "`, these procedure return a pipe port. Consequently, it is not possible to open a file whose name starts with those two characters.

`(open-input-string` *string*`)` procedure

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

`(open-output-string)` procedure

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

`(get-output-string` *port*`)` 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*`)` procedure
`(close-output-port` *port*`)` procedure

Identical to R^4RS.

`(read)` procedure
`(read` *port*`)` procedure
`(read-char)` procedure
`(read-char` *port*`)` procedure
`(peek-char)` procedure
`(peek-char` *port*`)` procedure
`(char-ready?)` procedure
`(char-ready?` *port*`)` procedure

Identical to R^4RS.

`(read-line)` procedure
`(read-line` *port*`)` procedure

Reads the next line available from the input port *port* and returns it as a string. The terminating newline is not included in the string. 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`.

`(write` *obj*`)` procedure
`(write` *obj port*`)` procedure
`(display` *obj*`)` procedure
`(display` *obj port*`)` procedure
`(newline)` procedure
`(newline` *port*`)` procedure
`(write-char` *char*`)` procedure
`(write-char` *char port*`)` procedure

Identical to R^4RS.

`(format` *port string* obj_1 obj_2 ...`)` procedure

Writes the *obj*s to the given *port*, according to the format string *string*. *String* 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*.
- ~~ is replaced by a single tilde.
- ~% is replaced by a newline

Port can be a boolean, a port or a string 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\"."
```

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

Returns the string associated with the output string *port*.

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

`(flush)` procedure
`(flush` *port*`)` procedure

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

`(load` *filename*`)` procedure

Identical to R^4RS.
Note: The *load* primitive has been extended to allow loading of object files, though this is not implemented on all systems [2]. See [**?**] for more details.

`(try-load` *filename*`)` procedure

Tries to load the file named *filename*. If *filename* exists and is readable, it is loaded, and `try-load` returns `#t`. Otherwise, the result of the call is `#f`.

`(autoload` *filename* ⟨symbol$_1$⟩ ⟨symbol$_2$⟩ ...`)` syntax

Defines ⟨symbol⟩s as autoload symbols associated to file *filename*. Fisrt evaluation of an autoload symbol will cause the loading of its associated file. *Filename* must provide a definition for the symbol which lead to its loading, otherwise an error is signaled.

`(autoload?` *symbol*`)` procedure

Returns `#t` if *symbol* is an autoload symbol; returns `#f` otherwise.

`(require` *string*`)` procedure
`(provide` *string*`)` procedure
`(provided?` *string*`)` 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.

`(transcript-on` *filename*`)` procedure
`(transcript-off)` procedure

Not implemented.

[2] Current version (3.0) allows image dumping only on some platforms:SunOs 4.1.x, SunOs 5.3, NetBSD 1.0, HPUX, Irix 5.3

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

Opens the file whose name is *filename* with the specified *mode*. *Mode* must be "r" to open for reading or "w" to open for writing. 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 `"| ``` permit to open a pipe port.

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

Closes *port*. If *port* denotes a string port, further reading or writing on this port is disallowed.

`(transcript-on` *filename*`)` procedure
`(transcript-off)` procedure

Not implemented.

`(port->string` *port*`)` procedure
`(port->list` *reader port*`)` procedure
`(port->string-list` *port*`)` procedure
`(port->sexp-list` *port*`)` procedure

Those procedures are utility for generally parsing input streams. Their specification has been stolen from `scsh`.
`Port->string` reads the input port until eof, then returns the accumulated string.

```
(port->string (open-input-file "| (echo AAA; echo BBB)"))
                              ⟹ "AAA\nBBB\n"
(define exec
        (lambda (command)
           (call-with-input-file
              (string-append "| " command) port->string)))

(exec "ls -l")              ⟹ a string which contains the result of "ls -l"
```

`Port->list` uses the *reader* function to repeatedly read objects from *port*. Thes objects are accumulated in a list which is returned upon eof.

```
(port->list read-line (open-input-file "| (echo AAA; echo BBB)"))
                              ⟹ ("AAA" "BBB")
```

`Port->string-list` reads the input port line by line until eof, then returns the accumulated list of lines. This procedure is defined as

```
(define port->string-list (lambda (p)(port->list read-line p)))
```

`Port->sexp-list` repeatedly reads data from the port until eof, then returns the accumulated list of items. This procedure is defined as

```
(define port->sexp-list (lambda (p) (port->list read p)))
```

For instance, the following expression gives the list of users currently connected on the machine running the STk interpreter.

```
(port->sexp-list (open-input-file "| users"))
```

6.11 Keywords

Keywords are symbolic constants which evaluate to themselves. A keyword must begin with a colon.

`(keyword?` *obj*`)` procedure

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

`(make-keyword` *obj*`)` procedure

Builds a keyword from the given *obj*. *obj* must be a symbol or a string. A colon is automatically prepended.

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

`(keyword->string` *keyword*`)` procedure

Returns the name of *keyword* as a string. The leading colon is included in the result.

```
(keyword->string :test)
          ⟹ ":test"
```

`(get-keyword` *keyword list*`)` procedure
`(get-keyword` *keyword list default*`)` procedure

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

```
(get-keyword :one '(:one 1 :two 2))
          ⟹ 1
(get-keyword :four '(:one 1 :two 2) #f)
          ⟹ #f
(get-keyword :four '(:one 1 :two 2))
          ⟹ error
```

6.12 Tk commands

As we mentioned in the introduction, STk can easily communicate with the Tk toolkit. All the commands defined by the Tk toolkit are visible as `Tk-command`s, a basic type recognized by the interpreter. `Tk-command`s can be called like regular scheme procedures, serving as an entry point into the Tk library.

Note: Some `Tk-command`s can dynamically create other `Tk-command`s. For instance, execution of the expression

```
(label '.lab)
```

will create a new `Tk-command` called "*.lab*". This new object, which was created by a primitive `Tk-command`, will be called a *widget*.
Note: When a new widget is created, it captures its creation environment. This permits to have bindings which access variables in the scope of the widget creation call (see 6.16).

`(tk-command?` *obj*`)` procedure

Returns `#t` if *obj* is a `Tk-command`, otherwise returns `#f`.

```
(tk-command? label)
          ⟹ #t
(begin (label '.lab) (tk-command? .lab))
          ⟹ #t
(tk-command? 12)
          ⟹ #f
```

`(widget?` *obj*`)` procedure

Returns `#t` if *obj* is a widget, otherwise returns `#f`. A widget is a `Tk-command` created by a primitive `Tk-command` such as `button`, `label`, `menu`, etc.

```
(widget? label)
          ⟹ #f
(begin (label '.lab) (widget? .lab))
          ⟹ #t
(widget? 12)
          ⟹ #f
```

`(widget->string` *widget*`)` procedure

Returns the widget name of *widget* as a string.

```
(begin (label '.lab) (widget->string .lab))
          ⟹ ".lab"
```

`(string->widget` *str*`)` procedure

Returns the widget whose name is *str* if it exists; otherwise returns `#f`.

```
(begin (label '.lab) (string->widget ".lab"))
          ⟹ the Tk-command named ".lab"
```

`(widget-name` *widget*`)` procedure

Returns the widget name of *widget* as a symbol.

```
(begin (label '.lab) (widget->name .lab))
          ⟹ .lab
```

`(set-widget-data!` *widget expr*`)` procedure

`Set-widget-data!` associates arbitrary data with a *widget*. The system makes no assumptions about the type of `expr`; the data is for programmer convenience only. As shown below, it could be used as a kind of property list for widgets.

`(get-widget-data` *widget*`)` procedure

Returns the data previously associated with *widget* if it exists; otherwise returns `#f`.

```
(begin
   (set-widget-data! .w '(:mapped #t :geometry "10x50"))
   (get-keyword :mapped (get-widget-data .w)))
         ⟹ #t
```

6.13 Environments

Environments are first class objects in STk. The following primitives are defined on environments.

`(environment?` *obj*`)` procedure

Returns `#t` if *obj* is an environment, otherwise returns `#f`.

`(the-environment)` procedure

Returns the current environment.

`(global-environment)` procedure

Returns the "global" environment (i.e. the toplevel environment).

`(parent-environment` *env*`)` procedure

Returns the parent environment of *env*. If env is the "global" environment (i.e. the toplevel environment), `parent-environment` returns `#f`.

`(environment->list` *environment*`)` procedure

Returns a list of *a-lists*, representing the bindings in *environment*. Each *a-list* describes one level of bindings, with the innermost level coming first.

```
(define E (let ((a 1) (b 2))
            (let ((c 3))
              (the-environment))))

(car (environment->list E)) ⟹ ((c . 3))

(cadr (environment->list E))⟹ ((b . 2) (a . 1))
```

`(procedure-environment` *procedure*`)` procedure

Returns the environment associated with *procedure*. `Procedure-environment` returns `#f` if *procedure* is not a closure.

```
(define foo (let ((a 1)) (lambda () a)))
(car (environment->list
         (procedure-environment foo)))
                          ⟹ ((a . 1))
```

`(symbol-bound?` *symbol*`)` procedure
`(symbol-bound?` *symbol environment*`)` procedure

Returns `#t` if *symbol* has a value in the given *environment*, otherwise returns `#f`. *Environment* may be omitted, in which case it defaults to the global environment.

6.14 Macros

STk provides low level macros.
Note: STk macros are not the sort of macros defined in the appendix of R^4RS, but rather the macros one can find in most of Lisp dialects.

`(macro` ⟨formals⟩ ⟨body⟩`)` syntax

`Macro` permits to create a macro. When a macro is called, the whole form (i.e. the macro itself and its parameters) is passed to the macro body. Binding association is done in the environment of the call. The result of the binding association is called the *macro-expansion*. The result of the macro call is the result of the evaluation of the macro expansion in the call environment.

```
(define foo (macro f `(quote ,f)))
(foo 1 2 3)                 ⟹ (foo 1 2 3)

(define 1+ (macro form (list + (cadr form) 1)))
(let ((x 1)) (1+ x))        ⟹ 2
```

`(macro?` *obj*`)` procedure

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

`(macro-expand-1` *form*`)` procedure
`(macro-expand` *form*`)` procedure

`Macro-expand-1` returns the macro expansion of *form* if it is a macro call, otherwise *form* is returned unchanged. `Macro-expand` is similar to `macro-expand-1`, but repeatedly expand *form* until it is no longer a macro call.

```
(define 1- (macro form `(-  ,(cadr form) 1)))
(define -- (macro form `(1- ,(cadr form))))
(macro-expand-1 '(1- 10))   ⟹ (- 10 1)
(macro-expand   '(1- 10))   ⟹ (- 10 1)
(macro-expand-1 '(-- 10))   ⟹ (1- 10)
(macro-expand   '(-- 10))   ⟹ (- 10 1)
```

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

Returns the macro expansion of *form* if it is a macro call, otherwise *form* is returned unchanged. Macro expansion continue until, the form obtained is

```
(define 1- (macro form (list '- (cadr form) 1)))
(macro-expand '(1- 10))     ⟹ (- 10 1)
```

`(macro-body` *macro*`)` procedure

Returns the body of *macro*

```
(macro-body 1+)
                              ⟹ (macro form (list + (cadr form) 1))
```

`(define-macro` *(⟨name⟩ ⟨formals⟩)* ⟨body⟩`)` macro

`Define-macro` is a macro which permits to define a macro more easily than with the `macro` form. It is similar to the `defmacro` of Common Lisp [4].

```
(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))
```

Note: Calls to macros defined by `define-macro` are physically replaced by their macro-expansion if the variable `*debug*` is `#f` (i.e. their body is "in-lined" in the macro call). To avoid this feature, and to ease debugging, you have to set this variable to `#t`. (See also 6.22).

6.15 System procedures

This section lists a set of procedures which permits to access some system internals.

`(expand-file-name` *string*`)` procedure

`Expand-file-name` expands the filename given in *string* to an absolute path. This function understands the *tilde convention* for filenames.

```
;; Current directory is /users/eg/STk
(expand-file-name "..")
          ⟹ "/users/eg"
(expand-file-name "~root/bin)
          ⟹ "/bin"
(expand-file-name "~/STk)"
          ⟹ "/users/eg/STk"
```

`(canonical-path` *path*`)` procedure

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

`(dirname` *string*`)` procedure

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

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

`(basename` *string*`)` procedure

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

```
(basname "/a/b/c.stk")
          ⟹ "c.stk"
```

`(decompose-file-name` *string*`)` 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")
```

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

`(glob` $pattern_1$ $pattern_2$ `...)` procedure

The code for `glob` is taken from the Tcl library. It 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 the *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.

`(getcwd` *string*`)` procedure

`Getcwd` returns a string containing the current working directory.

`(chdir` *string*`)` procedure

`Chdir` changes the current directory to the directory given in *string*.

`(getpid` *string*`)` procedure

Returns the system process number of the current STk interpreter (i.e. the Unix *pid*). Result is an integer.

`(system` *string*`)` procedure
`(!` *string*`)` procedure

Sends the given *string* to the system shell */bin/sh*. The result of `system` is the integer status code the shell returns.

`(exec` *string*`)` procedure

Executes the command contained in *string* and redirects its output in a string. This string constitutes the result of `exec`.

`(getenv` *string*`)` procedure

Looks for the environment variable named *string* and returns its value as a string, if it exists. Otherwise, `getenv` returns `#f`.

```
(getenv "SHELL")
          ⟹ "/bin/zsh"
```

6.16 Addresses

An *address* is a Scheme object which contains a reference to another Scheme object. This type can be viewed as a kind of pointer to a Scheme object. Addresses, even though they are very dangerous, have been introduced in STk so that objects that have no "readable" external representation can still be transformed into strings and back without loss of information. Adresses were useful with pre-3.0 version of STk; their usage is now **stongly discouraged**, unless you know what you do. In particular, an address can designate an object at a time and another one later (i.e. after the garbage collector has marked the zone as free).
Addresses are printed with a special syntax: `#pNNN`, where `NNN` is an hexadecimal value. Reading this value back yields the original object whose location is `NNN`.

`(address-of` *obj*`)` procedure

Returns the address of `obj`.

`(address?` *obj*`)` procedure

Returns `#t` if *obj* is an address; returns `#f` otherwise.

6.17 Signals

STk allows the use to associate handlers to signals. Signal handlers for a given signal can even be chained in a list. When a signal occurs, the first signal of the list is executed. Unless this signal yields the symbol `break` the next signal of the list is evaluated. When a signal handler is called, the integer value of this signal is passed to it as (the only) parameter.
The following POXIX.1 constants for signal numbers are defined: `SIGABRT`, `SIGALRM`, `SIGFPE`, `SIGHUP`, `SIGILL`, `SIGINT`, `SIGKILL`, `SIGPIPE`, `SIGQUIT`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGCHLD`, `SIGCONT`, `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`. Moreover, the following constants,

which are often available on most systems are also defined[3]: `SIGTRAP`, `SIGIOT`, `SIGEMT`, `SIGBUS`, `SIGSYS`, `SIGURG`, `SIGCLD`, `SIGIO`, `SIGPOLL`, `SIGXCPU`, `SIGXFSZ`, `SIGVTALRM`, `SIGPROF`, `SIGWINCH`, `SIGLOST`.
See your Unix documentation for the exact meaning of each constant or [5]. Use symbolic constants rather than their numeric value if you plan to port your program on another system.
A special signal, managed by the interpreter, is also defined: `SIGHADGC`. This signal is raised when the garbage collector phase terminates.
When the interpreter starts running, all signals are sets to their default value, excepted `SIGINT` (generally bound to Control-C) which is handled specially.

`(set-signal-handler!` *sig handler*`)` procedure

Replace the handler for signal *sig* with *handler*. Handler can be

- `#t` to reset the signal handler for *sig* to the default system handler.
- `#f` to completly ignore *sig* (Note that Posix.1 states that `SIGKILL` and `SIGSTOP` cannot be caught or ignored).
- a one parameter procedure.

This procedure returns the new handler, or (length 1) handler list, associated to *sig*.

```
(let* ((x       #f)
       (handler (lambda (i) (set! x #t))))
  (set-signal-handler! |SIGHADGC| handler)
  (gc)
  x)
        ⟹ #t
```

`(add-signal-handler!` *sig handler*`)` procedure

Adds *handler* to the list of handlers for signal *sig*. If the old signal handler is a boolean, this procedure is equivalent to `set-signal-handler!`. Otherwise, the new handler is added in front of the previous list of handler. This procedure returns the new handler, or handler list, associated to *sig*.

```
(let* ((x        '())
       (handler1 (lambda (i) (set! x (cons 1 x))))
       (handler2 (lambda (i) (set! x (cons 2 x)))))
  (add-signal-handler! |SIGHADGC| handler1)
  (add-signal-handler! |SIGHADGC| handler2)
  (gc)
  x)
        ⟹ (1 2)


(let* ((x        '())
       (handler1 (lambda (i) (set! x (cons 1 x))))
       (handler2 (lambda (i) (set! x (cons 2 x)) 'break)))
  (add-signal-handler! |SIGHADGC| handler1)
  (add-signal-handler! |SIGHADGC| handler2)
  (gc)
  x)
        ⟹ (2)
```

[3]Some of these constants may be undefined if they are not supported by your system

`(get-signal-handlers)` procedure
`(get-signal-handlers` *sig*`)` procedure

Returns the handlers, or the list of handlers, associated to the signal *sig*. If *sig* is omitted, `get-signal-handlers` returns a vector of all the signal handlers currently in effect.

6.18 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.
STk 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.

Note: Hash table manipulation procedures are built upon the efficient Tcl hash table package.

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

`Make-hash-table` admits three different forms. The most general form admit two arguments. The first argument is a comparison function which determine 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.
- `comparison` defaults to the `eq?` procedure

Consequently,

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

is equivalent to

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

Another 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` fucntion to speed up hash table retrievals).

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

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

`(hash-table-hash` *obj*`)` procedure

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

`(equal? x y)` implies `(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*`)` procedure

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

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

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

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

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

`(hash-table-remove!` *hash key*`)` procedure

hash must be a hash table containing an entry for *key*. `Hash-table-remove!` deletes the entry for *key* in *hash*, if it exists. Result of `Hash-table-remove!` is unspecified.

```
(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-for-each` *hash proc*`)` 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 undefined.

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*`)` procedure

Proc must be a procedure taking two arguments. `Hash-table-map` calls *proc* on each entry in *hash*, with the entry's key as the first argument and the entry's value as the second. The result of `hash-table-map` is a list of the values returned by *proc*, in 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))))
        ⟹ ((0 . "0") (3 . "3") (2 . "2") (1 . "1") (4 . "4"))
```

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

`hash-table->list` 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))
        ⟹ ((0 . "0") (3 . "3") (2 . "2") (1 . "1") (4 . "4"))
```

`(hash-table-stats` *hash*`)` procedure

`Hash-table-stats` returns a string with 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.

6.19 Regular expressions

Regular expressions are first class objects in STk. A regular expression is created by the `string->regexp` procedure. Matching a regular expression against a string is simply done by applying a previously created regular expression to this string. Regular expressions are implemented using code in the

Henry Spencer's package, and much of the description of regular expressions below is copied from his manual.

`(string->regexp` *string*`)` procedure

`String->regexp` compiles the *string* and returns the corresponding regular expression.

Matching a regular expression against a string is done by applying the result of `string->regexp` to this string. This application yields a list of integer couples if a matching occurs; it returns `#f` otherwise. Those integers correspond to indexes in the string which match the regular expression.
A regular expression is zero or more *branches*, separated by "`|`". It matches anything that matches one of the branches.
A branch is zero or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.
A piece is an *atom* possibly followed by "`*`", "`+`", or "`?`". An atom followed by "`*`" matches a sequence of 0 or more matches of the atom. An atom followed by "`+`" matches a sequence of 1 or more matches of the atom. An atom followed by "`?`" matches a match of the atom, or the null string.
An atom is a regular expression in parentheses (matching a match for the regular expression), a *range* (see below), "`.`" (matching any single character), "`^`" (matching the null string at the beginning of the input string), "`$`" (matching the null string at the end of the input string), a "`\`" followed by a single character (matching that character), or a single character with no other significance (matching that character).
A *range* is a sequence of characters enclosed in "`[]`". It normally matches any single character from the sequence. If the sequence begins with "`^`", it matches any single character *not* from the rest of the sequence. If two characters in the sequence are separated by "`-`", this is shorthand for the full list of ASCII characters between them (e.g. "`[0-9]`" matches any decimal digit). To include a literal "`]`" in the sequence, make it the first character (following a possible "`^`"). To include a literal "`-`", make it the first or last character.
In general there may be more than one way to match a regular expression to an input string. Considering only the rules given so far could lead to ambiguities. To resolve those ambiguities, the generated regular expression chooses among alternatives using the rule "first then longest". In other words, it considers the possible matches in order working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

1. If a regular expression could match two different parts of an input string then it will match the one that begins earliest.

2. If a regular expression contains "`|`" operators then the leftmost matching sub-expression is chosen.

3. In "`*`", "`+`", and "`?`" constructs, longer matches are chosen in preference to shorter ones.

4. In sequences of expression components the components are considered from left to right.

```
(define r1 (string->regexp "abc"))
(r1 "xyz")                  ⟹ #f
(r1 "12abc345")             ⟹ ((2 5))
(define r2 (string->regexp "[a-z]+"))
(r2 "12abc345")             ⟹ ((2 5))
```

If the regular expression contains parenthesis, and if there is a match, the result returned by the application will contain several couples of integers. First couple will be the indexes of the first longest substring which match the regular expression. Subsequent couples, will be the indexes of all the sub-parts of this regular expression, in sequence.

```
(define r3 (string->regexp "(a*)(b*)c"))
(r3 "abc")                  ⟹ ((0 3) (0 1) (1 2))
(r3 "c")                    ⟹ ((0 1) (0 0) (0 0))
((string->regexp "([a-z]+),([a-z]+)") "XXabcd,eXX")
                            ⟹ ((2 8) (2 6) (7 8))
```

`(regexp?` *obj*`)` procedure

Returns `#t` if *obj* is a regular expression created by `string->regexp`; otherwise returns `#f`.

```
(regexp? (string->regexp "[a-zA-Z][a-zA-Z0-9]*"))
                            ⟹  #t
```

`(regexp-replace` *pattern string substitution*`)` procedure
`(regexp-replace-all` *pattern string substitution*`)` procedure

`Regexp-replace` matches the regular expression *pattern* against *string*. If there is a match, the portion of *string* which match *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 strings of the form "`\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 "a*b" "aaabbcccc" "X")
                            ⟹ "Xbcccc"
(regexp-replace (string->regexp "a*b") "aaabbcccc" "X")
                            ⟹ "Xbcccc"
(regexp-replace "(a*)b" "aaabbcccc" "X\\1Y")
                            ⟹ "XaaaYbcccc"
(regexp-replace "(a*)b" "aaabbcccc" "X\\0Y")
                            ⟹ "XaaabYbcccc"
(regexp-replace "([a-z]*) ([a-z]*)" "john brown" "\\2 \\1")
                            ⟹ "brown john"
```

`Regexp-replace` replaces the first occurence of *pattern* in *string*. To replace *all* the occurences of the *pattern*, use `regexp-replace-all`

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

6.20 Processes

STk provides access to Unix processes as first class objects. Basically, a process contains four informations: the standard Unix process identification (aka PID) and the three standard files of the process.

`(run-process` *command* p_1 p_2 p_3 ...`)` 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.

The following example launches a process which execute 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" :wait #f)
```

`(process?` *process*`)` procedure

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

`(process-alive?` *process*`)` procedure

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

`(process-pid` *process*`)` procedure

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

`(process-input` *process*`)` procedure
`(process-output` *process*`)` procedure
`(process-error` *process*`)` procedure

Returns the file port associated to the standard input, output or error of *process*, 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` *process*`)` procedure

`Process-wait` stops the current process until *process* completion. `Process-wait` returns `#f` when *process* is already terminated; it returns `#t` otherwise.

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

`Process-exit-status` returns the exit status of *process* if it has finished its execution; returns `#f` otherwise.

`(process-send-signal` *process n*`)` procedure

Send the signal whose integer value is *n* to *process*. Value of *n* is system dependant. Use the defined signal constants to make your program indpendant of the running system (see 6.17). The result of *process-send-signal* is undefined.

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

`Process-kill` brutally kills *process*. The result of `process-kill` is undefined. This procedure is equivalent to

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

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

Those procedures are only available on systems which support job control. *Process-stop* stops the execution of *process* and *process-continue* resumes its execution. They are equivalent to

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

`(process-list)` procedure

`process-list` returns the list of processes which are currently running (i.e. alive).

6.21 Sockets

STk 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*`)` procedure

`make-client-socket` returns a new socket object. This socket establishes a link between the running application listening on port *port-number* of *hostname*.

`(socket?` *socket*`)` procedure

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

`(socket-hostname` *socket*`)` procedure

Returns a string which contains the name of the host on which *socket* is connected. This name is always the string "`localhost`" if *socket* is a server socket, it is the host name given when `make-client-socket` was called otherwise.

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

Returns the integer number of the port used for the listening *socket*.

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

Returns the file port associated for reading or writing with the program connected with *socket*. If no connection has already been established, these fucntions returns `#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`"[4]:

[4]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.

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

`(make-server-socket)` procedure
`(make-server-socket` *port-number*`)` 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 choosen by the system.

`(socket-accept-connection` *socket*`)` procedure

`socket-accept-connection` 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 to *socket*. This procedure must be called on a server socket created with `make-server-socket`. The result of `socket-accept-connection` is undefined.
The following exemple is a simple server which waits for a connection on the port 1234[5]. 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 ((s (make-server-socket 1234)))
  (socket-accept-connection s)
  (let ((l (read-line (socket-input s))))
    (format (socket-output s) "Length is: ~A\n" (string-length l))
    (flush (socket-output s)))
  (socket-shutdown s))
```

`(socket-shutdown` *socket*`)` procedure
`(socket-shutdown` *socket close*`)` procedure

`Socket-shutdown` shutdowns the connection associated to *socket*. *Close* is a boolean; it indicates if the socket must be close or not, when the connection is dstroyed. Closing the socket forbids further connections on the same port with the `socket-accept-connection` procedure. Omitting a value for *close* implies the closing of socket. The result of `socket-shutdown` is undefined.
The following example shows a simple server: when there is a new connection on the port number 1234, 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 1234)))
  (let loop ()
    (socket-accept-connection s)
    (format #t "I've read: ~A\n" (read-line (socket-input s)))
    (socket-shutdown s #f)
    (loop)))
```

[5] Under Unix, you can simply connect to listening socket with the `telnet` command. With the given example, this can be achived by typing the following command in a window shell:
`$ telnet localhost 1234`

6.22 Miscellaneous

This section lists the primitives defined in STk that did not fit anywhere else.

`(eval` ⟨expr⟩`)` syntax
`(eval` ⟨expr⟩ ⟨environment⟩`)` syntax

Evaluates ⟨expr⟩ in the given environment. ⟨Environment⟩ may be omitted, in which case it defaults to the global environment.

```
(define foo (let ((a 1)) (lambda () a)))
(foo)                        ⟹ 1
(eval '(set! a 2) (procedure-environment foo))
(foo)                        ⟹ 2
```

`(version)` procedure

returns a string identifying the current version of STk.

`(machine-type)` procedure

returns a string identifying the kind of machine which is running the interpreter. The form of the result is `[os-name]-[os-version]-[processor-type]`.

`(random` *n*`)` procedure

returns an integer in the range 0, $n - 1$ inclusive.

`(set-random-seed!` *seed*`)` procedure

Set the random seed to the specified *seed*. `Seed` must be an integer which fits in a C `long int`.

`(eval-string` *string environment*`)` procedure

Evaluates the contents of the given *string* in the given *environment* and returns its result. If *environment* is omitted it defaults to the global environment. If evaluation leads to an error, the result of `eval-string` is undefined.

```
(define x 1)
(eval-string "(+ x 1)")
          ⟹ 2
(eval-string "x" (let ((x 2)) (the-environment)))
          ⟹ 2
```

`(read-from-string` ⟨string⟩`)` procedure

Performs a read from the given *string*. If *string* is the empty string, an end of file object is returned. If an error occurs during string reading, the result of `read-from-string` is undefined.

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

`(dump` *string*`)` procedure

`Dump` grabs the current continuation and creates an image of the current STk interpreter in the file whose name is *string*[6]. This image can be used later to restart the interpreter from the saved state. See the STk man page about the `-image` option for more details.
Note: Image creation cannot be done if Tk is initialized.

`(trace-var` *symbol thunk*`)` procedure

`Trace-var` call the given *thunk* when the value of the variable denoted by *symbol* is changed.

```
(define x 1)
(define y 0)
(trace-var 'x (lambda () (set! y 1)))
(set! x 2)
(cons x y)
          ⟹ (2 . 1)
```

Note: Several traces can be associated with a single symbol. They are executed in reverse order to their definition. For instance, the execution of

```
(begin
  (trace-var 'z (lambda () (display "One")))
  (trace-var 'z (lambda () (display "Two")))
  (set! z 10))
```

will display the string `"Two"` before the string `"One"` on the current output port.

`(untrace-var` *symbol*`)` procedure

Deletes all the traces associated to the variable denoted by *symbol*.

`(error` *string* $string_1$ obj_2 ...`)` procedure

`error` prints the *obj*s according to the specification given in *string* on the current error port (or in an error window if Tk is initialized). The specification string follows the "tilde conventions" of `format`(see 6.10). Once the message is printed, execution returns to toplevel.

`(gc)` procedure

Runs the garbage collector. See 6.17 for the signals associated to garbage collection.

`(gc-stats)` procedure

Provides some statistics about current memory usage. This procedure is primarily for debugging the STk interpreter, hence its weird printing format.

`(expand-heap` *n*`)` procedure

Expand the heap so that it will contains at least *n* cells. Normally, the heap automatically grows when more memory is needed. However, using only automatic heap growing is sometimes very penalizing. This is particulary true for programs which uses a lot of temporary data (which are not pointed by any a variable) and a small amount of global data. In this case, the garbage collector will be often called and the heap will not be automatically expanded (since most of the consumed heap will be reclaimed by the GC). This could be annoying epecially for program where response

[6]Image creation is not yet implemented on all systems. The current version (3.0) allows image dumping only on some platforms: SunOs 4.1.x, Linux 1, FreeBsd

time is critical. Using `expand-heap` permits to enlarge the heap size (which is set to 20000 cells by default), to avoid those continual calls to the GC.

`(get-internal-info)` procedure

Returns a 7-length vector which contains the following informations:

0 total cpu used in milli-seconds

1 number of cells currently in use.

2 total number of allocated cells

3 number of cells used since the last call to `get-internal-info`

4 number of gc runs

5 total time used in the gc

6 a boolean indicating if Tk is initialized

`(sort` *obj predicate*`)` 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")
```

`(uncode` *form*`)` procedure

When STk evaluates an expression it encodes it so that further evaluations of this expression will be more efficient. Since encoded forms are generally difficult to read, `uncode` can be used to (re-)obtain the original form.

```
(define (foo a b)
   (let ((x a) (y (+ b 1))) (cons x y)))

(procedure-body foo)
                              ⟹ (lambda (a b)
                           (let ((x a) (y (+ b 1))) (cons x y)))
(foo 1 2)                     ⟹ (1 . 3)
(procedure-body foo)
                              ⟹  (lambda (a b)
                           (#let (x y)
                                 (#<local a @0,0)>
                                  (#<global +> #<local b @0,1)> 1))
                             (#<global cons> #<local x @0,0)>
                                             #<local y @0,1)>)))

(uncode (procedure-body foo))
                              ⟹ (lambda (a b)
                            (let ((x a) (y (+ b 1))) (cons x y)))
```