STk Reference manual

Version 3.99

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

June 1998

Document Reference

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

Contents

I Reference Manual **5**

1 Overview of STk 7
2 Lexical conventions 7
2.1 Identifiers 7
2.2 Comments 7
2.3 Other notations 8
3 Basic concepts 8
4 Expressions 9
4.1 Primitive expression types 9
4.2 Derived expression types 9
5 Program structure 11
6 Standard procedures 11
6.1 Booleans 11
6.2 Equivalence predicates 11
6.3 Pairs and lists 12
6.4 Symbols 14
6.5 Numbers 15
6.6 Characters 17
6.7 Strings 18
6.8 Vectors 20
6.9 Control features 20
6.10 Input and output 21
6.11 Keywords 29
6.12 Tk commands 30
6.13 Modules 31
6.14 Environments 35
6.15 Macros 36
6.16 System procedures 37
6.17 Addresses 40
6.18 Signals 40
6.19 Hash tables 42
6.20 Regular expressions 45
6.21 Pattern matching 47
6.22 Processes 50
6.23 Sockets 52
6.24 Foreign Function Interface 55
6.25 Miscellaneous 59

II Annexes **67**

A Using the Tk toolkit **69**
1 Calling a Tk-command 69
2 Associating Callbacks to Tk-commands 70
3 Tk bindings 71

B Differences with R4RS **73**
1 Symbols 73
2 Types 75
3 Procedures 75

C An introduction to STklos **77**
1 Introduction 77
2 Class definition and instantiation 77
2.1 Class definition 77
3 Inheritance 78
3.1 Class hierarchy and inheritance of slots 78
3.2 Instance creation and slot access 79
3.3 Slot description 80
3.4 Class precedence list 83
4 Generic functions 84
4.1 Generic functions and methods 84
4.2 Next-method 85
4.3 Example 86

D Modules: Examples **89**

E Changes **93**

F Miscellaneous Informations **97**
1 Introduction 97
2 About STk 97
2.1 Latest release 97
2.2 Sharing Code 97
2.3 STk Mailing list 97
2.4 STk FAQ 98
2.5 Reporting a bug 98
3 STk and Emacs 98
3.1 Using the SLIB package with STk 99
4 Getting information about Scheme 99
4.1 The R^4RS document 99
4.2 The Scheme Repository 100
4.3 Usenet newsgroup and other addresses 100

Part I

Reference Manual

Introduction

This document provides a complete list of procedures and special forms implemented in version 3.99 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

There are three types of comments in STk:

1. a semicolon (;) indicates the start of a comment. This kind of comment extends to the end of the line (as described in R^4RS).

2. multi-lines comment use the classical Lisp convention: a comment begins with `#|` and ends with `|#`.

3. 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
```

[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`

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.

`#.<expr>` is read as the evaluation of the Scheme expression `<expr>`. The evaluation is done during the `read` process, when the `#.` is encountered. Evaluation is done in the environment of the current module.

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

`#n=` is used to represent circular structures . The value given of $n$miust be a number. It is used as a label, which can be referenced later by a `#n#` syntax (see below). The scope of the label is the expression being read by the outermost `read`.

`#n=` is used to reference a some object labeled by a `#n=` syntax; that is, `#n#` represents a pointer to the object labeled exactly by `#n=`. For instance, the object created returned by the following expression

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

caen be represented in this way:

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

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 6.15).

(`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? obj1 obj2)` 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? obj1 obj2)` procedure

Identical to R^4RS.

6.3 Pairs and lists

`(pair? obj)` procedure
`(cons obj1 obj2)` 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

Identical to R^4RS.

`(append! list ...)` procedure

Returns a list consisting of the elements of the first *list* followed by the elements of the other lists, as with **append**. The differenece with **append** is that the arguments are *changed* rather than *copied*.

```
(append! '(1 2) '(3 4) '(5 6))
       ⟹ '(1 2 3 4 5 6)
(let ((l1 '(1 2))
      (l2 '(3 4))
      (l3 '(5 6)))
  (append! l1 l2 l3)
  (list l1 l2 l3))
       ⟹ ((1 2 3 4 5 6) (3 4 5 6) (5 6))
```

`(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.

`(remq` *obj list*`)` procedure
`(remv` *obj list*`)` procedure
`(remove` *obj list*`)` procedure

Each function return a copy of *list* where all the occurences of *obj* have been deleted. The predicate used to test the presence of *obj* in *list* is respectively `eq`, `eqv` and `equal`.
Note: It is not an error if *obj* does not appear in *list*.

```
(remq 1 '(1 2 3))           ⟹ (2 3)
(remq "foo" '("foo" "bar")) ⟹ ("foo" "bar")
(remove "foo" '("foo" "bar"))
                            ⟹ ("bar")
```

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

Returns the last pair of *list*[2].

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

`(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.

[2] `Last-pair` was a standard procedure in R^3RS.

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 *R⁴RS* 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.

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

`(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.

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

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

(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.

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
```

`(split-string` *string*`)` procedure
`(split-string` *string delimiters*`)` procedure

This function parses *string*$_1$ and returns a list of tokens ended by a character of the *delimiters*$_1$ string. If *delimiters*$_1$ is omitted, it defaults to a string containing a space, a tabulation and a newline characters.

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

`(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^4RS.

(`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^4RS.

(`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` $\langle$thunk$_1\rangle$ $\langle$thunk$_2\rangle$ $\langle$thunk$_3\rangle$`)` procedure

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

`(catch` $\langle$expression$_1\rangle$ $\langle$expression$_2\rangle$ ...`)` syntax

The $\langle$expression$\rangle$s are evaluated from left to right. If an error occurs, evaluation of the $\langle$expression$\rangle$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` $\langle$procedure$\rangle$`)` procedure

returns the body of $\langle$procedure$\rangle$. If $\langle$procedure$\rangle$ 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 R^4RS 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 *string* 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 R^4RS.

`(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 R^4RS.

`(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 R^4RS.
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
a`(close-output-port` *port*`)` procedure

Identical to R^4RS.

`(read)` procedure
`(read` *port*`)` procedure

The STk procedure is identical to the R^4RSprocedure. It has bee extended to accept the "`#x=`" and "`#x#`" notations used for circular stuctures (see 2.3).

`(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

Identical to R^4RS.

`(write*` *obj*`)` procedure
`(write*` *obj port*`)` 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 "`#x=`" and "`#x#`" notations (see 2.3).
As `write`, the *port* argument can be omitted, defaulting to the value returned by `current-output-port`, and the value returned by `write*` is undefined.

```
(let ((l (cons 1 2)))
  (set-cdr! l l)
  (write* l))              ⟹ writes #0=(1 . #0#)

(let ((l1 '(1 2))
      (l2 '(3 4))
      (l3 '(5 6)))
  (append! l1 l2 l3)
  (list l1 l2 l3))         ⟹ writes ((1 2 . #0=(3 4 . #1=(5 6))) #0# #1#)
```

(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*.
- ~w or ~W is replaced by the printed representation of the next *obj* (circular structures are correctly handled and printed using `writes*`).
- ~~ is replaced by a single tilde.
- ~% 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`.

`(when-port-readable` *port handler*`)` procedure
`(when-port-readable` *port*`)` procedure

When *port* is ready for reading, *handler*, which must be a thunk, is called leaving the current evaluation suspended. When *handler* execution is terminated, normal evaluation is resumed at its suspension point. If the special value `#f` is provided as *handler*, the current handler for *port* is deleted. If a handler is provided, the value returned by `when-port-readable` is undefined. Otherwise, it returns the handler currently associated to *port*.
The example below shows a simple usage of the `when-port-readable` procedure: the command *cmd* is run with its output redirected in a pipe associated to the *p* Scheme port.

```
(define p (open-input-file "| cmd"))
(when-port-readable p
            (lambda()
              (let (( l (read-line p)))
                (if (eof-object? l)
                    (begin
                      ;; delete handler
                      (when-port-readable p #f)
                      ;; and close port
                      (close-input-port p))
                    (format #t "Line read: ~A\n" l)))))
```

`(when-port-writable` *port handler*`)` procedure
`(when-port-writable` *port*`)` procedure

When *port* is ready for writing, *handler*, which must be a thunk, is called leaving the current evaluation suspended. When *handler* execution is terminated, normal evaluation is resumed at its suspension point. If the special value `#f` is provided as *handler*, the current handler for *port* is deleted. If a handler is provided, the value returned by `when-port-writable` is undefined. Otherwise, it returns the handler currently associated to *port*.

`(load` *filename*`)` procedure
`(load` *filename module*`)` procedure

The first form is identical to R^4RS. The second one loads the content of *filename* in the given *module* environment. *Note:* The *load* primitive has been extended to allow loading of object files, though this is not implemented on all systems. This extension uses dynamic loading on systems which support it [3]. See [4] for more details.

`(try-load` *filename*`)` procedure
`(try-load` *filename module*`)` 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`. The second form of *try-load* tries to load the content of *filename* in the given *module* environment.

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

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

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

Returns `#t` if *symbol* is an autoload symbol in *module* environment ; 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.

`(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.

[3] Current version (3.99) allows dynamic loading only on some platforms: SunOs 4.1.x, SunOs 5.x, NetBSD 1.0, Linux 2.0, HPUX, Irix 5.3

`(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-commands`, a basic type recognized by the interpreter. `Tk-commands` can be called like regular scheme procedures, serving as an entry point into the Tk library.

Note: Some `Tk-commands` can dynamically create other `Tk-commands`. 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 6.17).

(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 Modules

STk 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.
STk provides a simple module system which is largely inspired from the one of Tung and Dybvig exposed in [5]. As their modules system, STk modules are defined to be easily used in an interactive environment.

`(define-module` *name* ⟨body⟩`)` syntax

Define-module evaluates the expressions which are in ⟨body⟩ in the environment of the module *name*. *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, ⟨body⟩ is evaluated in the environment of the (old) module *name*[4].
Definitions done in a module are local to the module and do not interact with the definitions of other modules. Consider the following definitions,

[4]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, inteactively reloading a module does not define a new entity, and the other modules which use it are not altered.

```
(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 distincts modules they denote two different locations.
The "`STk`" module, which is predefined, is a special module which contains all the *global variables* of a R^4RS program. A symbol defined in the `STk` 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 `STk` module.
The result of *define-module* is undefined.

(find-module *name***)** procedure
(find-module *name default***)** procedure

STk 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***)** procedure

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

```
(module? (find-module 'STk))
        ⟹ #t
(module? 'STk)
        ⟹ #f
(module? 1)
        ⟹ #f
```

(export ⟨symbol$_1$⟩ ⟨symbol$_2$⟩... **)** 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 the symbols which appear in an `export` clause.
If several `export` clauses appear in a module, the set of exported symbols is determined by *unioning* symbols exported in all the `export` clauses.
The result of *export* is undefined.

(import ⟨module$_1$⟩ ⟨module$_2$⟩... **)** 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 (⟨module$_1$⟩ ⟨module$_2$⟩...) which export 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)))
        ⟹ displays (m1-a m1-b m2-c)
```

Note: There is no kind of *transitivity* in module importations: when the module C imports the module B which an importer of A, the symbols of A are not visible from C, except by explicitly importing the A module from C. *Note:* The module `STk`, 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.

(export-symbol *symbol module***)** procedure

Exports *symbol* from *module*. This procedure can be useful, when debugging a program, to make visible a given symbol without reloading or redefining the module where this symbol was defined.

(export-all-symbols) procedure

Exports all the symbols of current module . If symbols are added to the current *module* after the call to `export-all-symbols`, they are automatically exported.
Note: The `STk` module export all the symbols which are defined in it (i.e. *global variables* are visible, if not hidden, from all the modules of a program.

(with-module *name* $\langle\text{expr}_1\rangle$ $\langle\text{expr}_2\rangle$ ...**)** syntax

Evaluates the expressions of $\langle\text{expr}_1\rangle$ $\langle\text{expr}_2\rangle$ ... in the environment of module *name*. Module *name* must have been created previously by a `define-module`. The result of `with-module` is the result of the evaluation of the last $\langle\text{expr}\rangle$.

```
(define-module M
  (define a 1)
  (define b 2))

(with-module M
   (+ a b))
        ⟹ 3
```

(current-module) procedure

Returns the current-module.

```
(define-module M
   ...)

 (with-module M
   (cons (eq? (current-module) (find-module 'M))
         (eq? (current-module) (find-module 'STk))))

         ⟹ (#t . #f)
```

`(select-module name)` syntax

Evaluates the expressions which follows in module *name* environment. Module *name* must have been created previously by a `define-module`. The result of `select-module` is undefined. `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`[5]:

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

`(module-name module)` procedure

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

`(module-imports module)` procedure

Returns the list modules that *module* imports.

`(module-exports module)` procedure

Returns the list of symbols exported by *module*.

`(module-symbols module)` procedure

Returns the list symbols that ere defined in *module*.

[5]This transcript uses the default value for the function `repl-display-prompt` (see page 74) which displays the name of the current module in the prompt.

6.14 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))
```

`(module-environment module)` procedure

Returns the environment associated to the module *module*.

```
(define-module M
   (define a 1))
(car (environment->list
         (module-environment (find-module 'M))))
                            ⟹ ((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.

6.15 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 repeately 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 [6].

```
(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 6.25).

6.16 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.

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

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

`(rename-file` $string_1$ $string_2$`)` procedure

Renames the file whose path-name is contained in $string_1$ in the path name given by $string_2$. The result of `rename-file` is undefined.

`(temporary-file-name` *string*`)` 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.

`(getcwd)` 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"
```

(setenv! *var value*) procedure

Sets the environment variable *var* to *value*. *Var* and *value* must be strings. The result of `setenv!` is undefined.

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

6.17 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.

6.18 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[6]: `SIGTRAP`, `SIGIOT`, `SIGEMT`,

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

`SIGBUS`, `SIGSYS`, `SIGURG`, `SIGCLD`, `SIGIO`, `SIGPOLL`, `SIGXCPU`, `SIGXFSZ`, `SIGVTALRM`, `SIGPROF`, `SIGWINCH`, `SIGLOST`.
See your Unix documentation for the exact meaning of each constant or [7]. 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)
```

`(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.

6.19 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.

6.20 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"
```

6.21 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; the STk "pattern matching compiler" provides this[7]. The technique used is described in details in [9], and the code generated can be considered optimal due to the way this "pattern compiler" was obtained.
The "pattern language" allows the expression of a wide variety of patterns, including:

[7] The "pattern matching compiler" has been written by Jean-Marie Geffroy and is part of the Manuel Serrano's Bigloo compiler[8] since several years. The code (and documentation) included in STk has been stolen from the Bigloo package v1.9 (the only difference between both package is the pattern matching of structures whisch is absent in STk).

- 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.

Pattern Matching Facilities

Only two special forms are provided for this: `match-case` and `match-lambda` and these also exist, for example, in Andrew Wright and Bruce Duba's [10] pattern matching package.

`(match-case` $\langle\text{key}\rangle$ $\langle\text{clause}_1\rangle$ $\langle\text{clause}_2\rangle\ldots$**`)`** syntax

In this form, $\langle\text{key}\rangle$ may be any expression and each $\langle\text{clause}\rangle$ has the form

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

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

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

`(match-lambda` $\langle\text{clause}_1\rangle$ $\langle\text{clause}_2\rangle\ldots$**`)`** syntax

The form `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)) 'bar)
        ⟹ bar
```

<table>
<tr><th><<i>pattern</i>> ⟶</th><th><i>Matches:</i></th></tr>
<tr><td><atom> || <code>(kwote</code> <atom><code>)</code></td><td>any expression <code>eq?</code> to <atom></td></tr>
<tr><td>|| <code>(and</code> <pat₁ > ... <pat_n ><code>)</code></td><td>if all of <pat_i > match</td></tr>
<tr><td>|| <code>(or</code> <pat> ... <pat_n ><code>)</code></td><td>if any of <pat₁ > through <pat_n >match</td></tr>
<tr><td>|| <code>(not</code> <pat><code>)</code></td><td>if <pat> doesn't match</td></tr>
<tr><td>|| <code>(?</code> <predicate><code>)</code></td><td>if <predicate> is true</td></tr>
<tr><td>|| <code>(</code><pat₁ > <code>...</code>[8] <pat_n ><code>)</code></td><td>a list of n elements</td></tr>
<tr><td>|| <pat> <u><code>...</code></u> [9]</td><td>a (possibly empty) repetition of <pat> in a list.</td></tr>
<tr><td>|| <code>#(</code><pat> <code>...</code><pat_n ><code>)</code></td><td>a vector of n elements</td></tr>
<tr><td>|| <code>?</code><identifier></td><td>anything, and binds identifier as a variable</td></tr>
<tr><td>|| <code>?-</code></td><td>anything</td></tr>
<tr><td>|| <code>??-</code></td><td>any (possibly empty) repetition of anything in a list</td></tr>
<tr><td>|| <code>???-</code></td><td>any end of list</td></tr>
</table>

Table 3: Pattern Syntax

The pattern language

The syntax is presented in Table 3. It is described below in the same way (and nearly in the same words) as in [10].

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

Explanations through examples

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

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

6.22 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 6.18). 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).

6.23 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-host-name` *socket*`)` 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 the socket, this function returns `#f`.

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

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

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

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

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

Returns the integer number of the port used for *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 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`"[10]:

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

[10]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.

`(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 chosen 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 example is a simple server which waits for a connection on the port 1234[11]. 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 closed or not, when the connection is destroyed. 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)))
```

`(socket-down?` *socket*`)` procedure

Returns `#t` if *socket* has been previously closed with `socket-shutdown`. It returns `#f` otherwise.

[11]Under Unix, you can simply connect to 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 1234`

`(socket-dup` *socket*`)` procedure

Returns a copy of *socket*. The original and the copy socket can be used interchangeably. However, if a new connection is accepted on one socket, the characters exchanged on this socket are not visible on the other socket. Duplicating a socket is useful when a server must accept multiple simultaneous connections. The following example creates a server listening on port 1234. This server is duplicated and, once two clients are present, a message is sent on both connections.

```
(define s1 (make-server-socket 1234))
(define s2 (socket-dup s1))
(socket-accept-connection s1)
(socket-accept-connection s2)
;; blocks until two clients are present
(display "Hello,\n" (socket-output s1))
(display "world\n"  (socket-output s2))
(flush (socket-output s1))
(flush (socket-output s2))
```

`(when-socket-ready` *socket handler*`)` procedure
`(when-socket-ready` *socket*`)` procedure

Defines a handler for *socket*. The handler is a thunk which is executed when a connection is available on *socket*. If the special value `#f` is provided as *handler*, the current handler for *socket* is deleted. If a handler is provided, the value returned by `when-socket-ready` is undefined. Otherwise, it returns the handler currently associated to *socket*.
This procedure, in conjunction with `socket-dup` permits to build multiple-clients servers which work asynchronously. Such a server is shown below.

```
(define p (make-server-socket 1234))
(when-socket-ready p
                  (let ((count 0))
                    (lambda ()
                      (set! count (+ count 1))
                      (register-connection (socket-dup p) count))))

(define register-connection
  (let ((sockets '()))
    (lambda (s cnt)
      ;; Accept connection
      (socket-accept-connection s)
      ;; Save socket somewhere to avoid GC problems
      (set! sockets (cons s sockets))
      ;; Create a handler for reading inputs from this new connection
      (let ((in  (socket-input s))
            (out (socket-output s)))
        (when-port-readable in
             (lambda ()
               (let ((l (read-line in)))
                 (if (eof-object? l)
                     ;; delete current handler
                     (when-port-readable in #f)
```

Name	**Corresponding C type**	**Corresponding Scheme Type**
`:void`	`void`	*None*
`:char`	`char`	Scheme character or Scheme integer
`:short`	`short int`	Scheme integer
`:ushort`	`unsigned short int`	Scheme integer
`:int`	`int`	Scheme integer
`:uint`	`unsigned integer`	Scheme integer
`:long`	`long integer`	Scheme integer
`:ulong`	`unsigned long integer`	Scheme integer
`:float`	`float`	Scheme Real
`:double`	`double`	Scheme Real
`:static-ptr`	pointer on a static area	Scheme C-pointer object or Scheme String
`:dynamic-ptr` *or* `(:void *)`	pointer on a dynamic area (`malloc`ated)	Scheme C-pointer object or Scheme String
`:string` *or* `(:char *)`	`char *` (pointer on a dynamic string)	Scheme C-pointer object or Scheme String
`:boolean`	`int`	Scheme boolean

Table 4: FFI predefined types

```
;; Just write the line read on the socket
(begin
  (format out "On #~A --> ~A\n" cnt l)
  (flush out)))))))))
```

6.24 Foreign Function Interface

The STk Foreign Function Interface (FFI for short) has been defined to allow an easy access to functions written in C without needing to build C-wrappers and, consequently, without any need to write C code. Note that the FFI is very machine dependent and that it works only on a limited set of architectures[12]. Moreover, since FFI allows very low level access, it is easy to crash the interpreter when using an external C function.

The definition of an external function is done with the syntax `define-external`. This form takes as arguments a typed list of parameters and accepts several options to define the name of the function in the C world, the library which defines this function, ... The type of the function result and the types of its arguments are defined in Table 4. This table lists the various keywords reserved for denoting types and their equivalence between the C and the Scheme worlds.

[12]In release 3.99, FFI is known to work on the following architectures : ix86 (but not yet MS Windows), Sun Sparc, HP 9000, SGI.

(define-external ⟨name⟩ ⟨parameters⟩ ⟨options⟩**)** syntax

The form `define-external` binds a new procedure to ⟨name⟩. The arity of this new procedure is defined by the typed list of parameters given by ⟨parameters⟩. This parameters list is a list of couples whose first element is the name of the parameter, and the second one is is a keyword representing its type (see table for equivalence). All the types defined in Table 4, except `:void`, are allowed for the parameters of a foreign function. `Define-external` accepts several options:

- `:return-type` is used to define the type of the value returned by the foreign function. The type returned must be chosen in the types specified in the table. For instance:

  ```
  (define-external maximum((a :int) (b :int))
    :return-type :int)
  ```

  defines the foreign function maximum which takes two C integers and returns an integer result. Omitting this option default to a result type equal to `:void` (i.e. the returned value is *undefined*).

- `:entry-name` is used to specify the name of the foreign function in the C world. If this option is omitted, the entry-name is supposed to be ⟨name⟩. For instance:

  ```
  (define-external minimum((a :int) (b :int))
    :return-type :int
    :entry-name  "min")
  ```

  defines the Scheme function `minimum` whose application executes the C function called `min`.

- `:library-name` is used to specify the library which contains the foreign-function. If necessary, the library is loaded before calling the C function. So,

  ```
  (define-external minimum((a :int) (b :int))
    :return-type  :int
    :entry-name   "min"
    :library-name "libminmax")
  ```

  defines a function which will execute the function `min` located in the library `libminmax.xx` (where `xx` is the suffix used for shared libraries on the running system (generally `so` or `sl`).

Hereafter, there are some commented definitions of external functions:

```
(define-external isatty ((fd :int))
   :return-type :boolean)

(define-external system ((cmd (:char *))) ;; or ((cmd :string))
   :return-type :int)

(define-external malloc ((size :ulong))
```

```
    :return-type (void *))

  (define-external free ( (p (:void *) )) )
```

All these functions are defined in the C standard library, hence it is not necessary to specify the `:library-name` option.

- `istty` is declared here as a function which takes an integer and returns a boolean (in fact, the value returned by the C function `isatty` is an `int`, but we ask here to the FFI system to translate this result as a boolean value in the Scheme world).
- `system` is a function which takes a string as parameter and returns an `int`. Note that the type of the parameter, can be specified as a `(:char *)` or `:string`, as indicated in Table 4.
- `malloc` is a function which takes one parameter (an `unsigned long int` and which returns a `(:void *)` (or `:dynamic-ptr`). Specifying that the result is a dynamic pointer (instead of a static one) means that we want that the Garbage Collector takes into account the area allocated by the C function `malloc` (i.e. if this area becomes no more accessible, the GC disposes it with the `free` function[13].
- `free` is a function which takes a dynamic pointer and deallocates the area it points. Since the definition of this function specifies no result type, it is supposed to be `:void`[14].

External functions can also have a variable number of parameters by using the standard Scheme *dot* notation. For instance,

```
(define-external printf ((format :string) . l)
  :return-type  :int)
```

defines a Scheme function with one or more parameters (the first one being a string). Of course, the parameters which constitute the variable parameters list must have a type which appears in the third column of Table 4. Some examples using the `printf` function:

```
(printf "This is a %s test" "good")

        ⟹ displays   This is a good test
(printf "char: '%c' Dec: '%04d' Hex '%04x'" #\space 100 100)

        ⟹ displays   char: ' ' Dec: '0100' Hex '0064'
```

Note: The types `:dynamic-ptr`, `:static-ptr` and `:string` are compatible when used for foreign function parameter. This gives a semantic which is similar to the one of C, where `void *` is a compatible with all other pointer types. However, differenciating those types is useful for converting the function return value to a proper Scheme type.

[13]Pointers defined with `:dynamic-ptr` are always unallocated with `free`. Consequently, areas allocated with another allocator than the standard one must be declared as `:static-ptr` and freed by hand

[14]Usage of malloc and free are for illustration purpose here. Their usage in a program must be avoided, if possible, because it can have interact badly with the way the interpreter manages memory or it can conduct to *crashing* programs if you don't take care.

Note: When a function has a `:return-type` which is `:string`, `:dynamic-ptr` or `:static-ptr`, and the return value is the C `NULL` pointer, the Scheme value returned by the function is, by convention, equal to `#f`. For instance, the GNU `readline` function allows line editing *à la* Emacs returns `NULL` when the user has typed an end of file. The following lines show how to make a simple shell-like toplevel using FFIs.

```
(define-external system ((var (:char *)))
  :return-type :int)

(define-external readline ((prompt :string))
  :library-name "libreadline"
  :return-type  :string)

;; A Shell-like toplevel
(do ((l (readline "?> ") (readline "?> ")))
    ((not l))
  (system l))
```

`(external-exists? `*entry*`)` procedure
`(external-exists? `*entry library*`)` procedure

Returns `#t` if *entry* is defined as an external symbol in *library*. If *library* is not provided the symbol is searched in the STk interpreter or in libraries that it uses. This function can be useful to define external functions conditionally:

```
(when (external-exists? "dup2")
   (define-external dup2 ((oldfd :int) (newfd :int))
      :return-type :int))
```

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

STk strings are more general than C strings since they accept null character. `c-string->string` takes an area of characters built by a call to a foreign function (typically the result of a function returning a `:static-ptr`, `:dynamic-ptr` or `:string`) and convert it to a proper Scheme string.

```
(define-external sprintf ((str :string) (format :string) . l)
   :return-type  :int)

(let ((str (make-string 5 #\?)))
  (sprintf str "%x" 100)
  (cons str (C-string->string str)))

        ⟹ ("64\0??" . "64")
```

6.25 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*[15]. 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 6.18 for the signals associated to garbage collection.

[15] Image creation is not yet implemented on all systems. The current version (3.99) allows image dumping only on some platforms: SunOs 4.1.x, Linux 1, FreeBsd

`(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 particularly 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 specially for program where response 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)))
```

Note: When a macro has been directly expanded into the macro call code, it is not possible to retrieve the original macro call. Set `*debug*` to `#t` to avoid macro expansion in-lining.

(time ⟨expr⟩**)** macro

Evaluates the expression ⟨expr⟩ in the current environment. Prints the elapsed CPU time and the number of conses used before returning the result of this evaluation.

(apropos *symbol***)** procedure

`Apropos` returns a list of symbol whose print name contains the characters of *symbol*. Symbols are searched for in the current environment.

```
(apropos 'cadd)
          ⟹ (caddar caddr cadddr)
```

(inspect *obj***)** procedure

`Inspect` permits to graphically inspect an object. The first call of this procedure creates a top level window containing the object to inspect and its current value. If the inspector window is already on screen, *obj* will be appended to the list of inspected objects. The inspector window contains menus which permit to call the viewer or detailer on each inspected object. See the on-line documentation for further details. A view of the general inspector is given in figure 1.
Note: Tk must be initialized to use `inspect`.

(view *obj***)** procedure

`View` permits to obtain a graphical representation of an STk object. The type of representation depends on the type of the viewed object. Here again, menus are provided to switch to

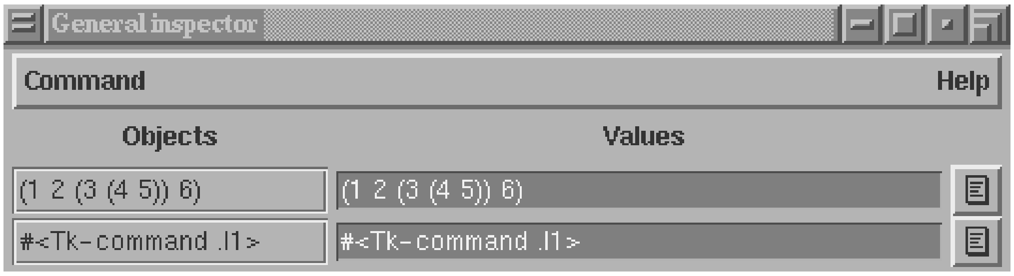

Figure 1: A view of the Inspector

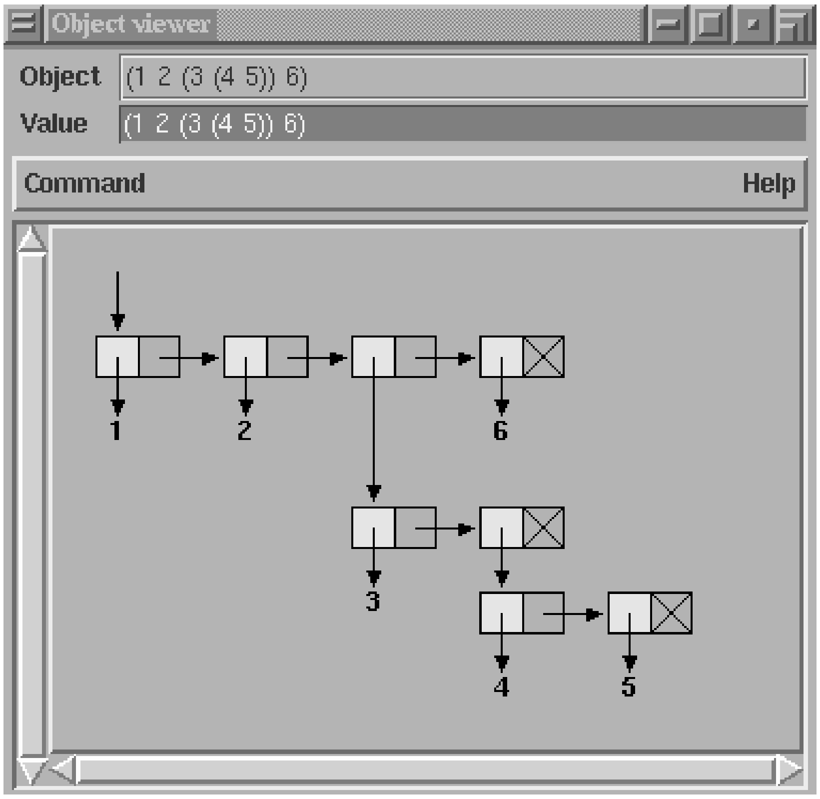

Figure 2: A view of the Viewer

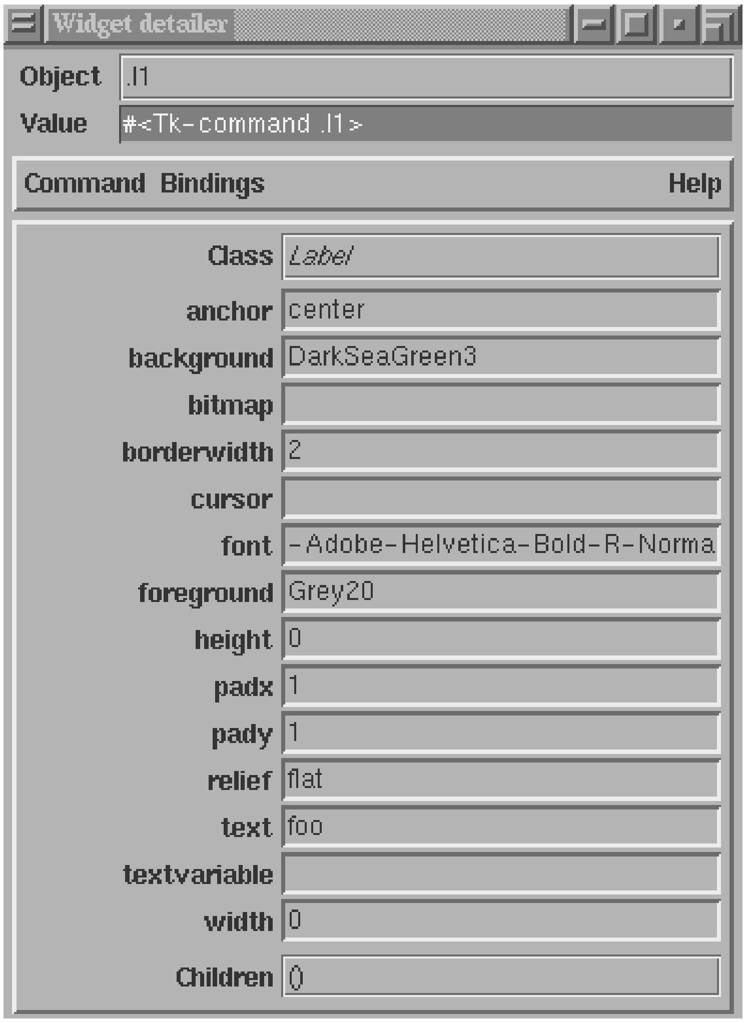

Figure 3: A view of the Detailer

the inspector or to the detailer. See the on-line documentation for more details. A snapshot of the viewer is given in figure 2.
Note: Tk must be initialized to use `view`.

`(detail` *obj*`)` procedure

`detail` permits to display the fields of a composite Scheme object. The type of detailer depends on the type of the composite object detailed. Here again, menus are provided to go to the inspector or to the viewer. See the on-line documentation for more details. Figure 3 shows the detailer examining a *tk-command*.
Note: Tk must be initialized to use `detail`.

`(quit` *retcode*`)` procedure
`(quit)` procedure
`(exit` *retcode*`)` procedure
`(exit)` procedure

`(bye` *retcode*`)` procedure
`(bye)` procedure

Exits the STk interpreter with the specified integer return code. If omitted, the interpreter terminates with a return code of 0.

Part II

Annexes

Appendix A

Using the Tk toolkit

When STk detects that a *tk-command* must be called, parameters are processed to be recognized by the corresponding toolkit function. Since the Tk toolkit is left (mostly) unmodified, all its primitives "think" there is a running Tcl interpreter behind the scene. Consequently, to work with the Tk toolkit, a little set of rewriting rules must be known. These rules are described hereafter.

Note: This appendix is placed here to permit an STk user to make programs with the original Tcl/Tk documentation by hand. In no case will it substitute to the abundant Tcl/Tk manual pages nor to the excellent book by J. Ousterhout[11]

1 Calling a Tk-command

Since Tcl uses strings to communicate with the Tk toolkit, parameters to a *Tk-command* must be translated to strings before calling the C function which implement it. The following conversions are done, depending on the type of the parameter that STk must give to the toolkit:

symbol: the print name of the symbol;

number: the external representation of the number expressed in radix 10;

string: no conversion;

keyword: the print name of the keyword where the initial semicolon has been replaced by a dash ("-");

boolean: the string "0" if `#f` and "1" if `#t`

tk-command: the name of the *tk-command*

closure: the address of the closure using the representation shown in 6.17.

otherwise: the external "slashified" version of the object.

As an example, let us make a button with a label containing the string `"Hello, word"`. According the original Tk/Tcl documentation, this can be done in Tcl with

```
button .hello -text "Hello, world"
```

Following the rewriting rules expressed above, this can be done in STk with

```
(button '.hello '-text "Hello, world")
```

This call defines a new widget object which is stored in the STk variable `.hello`. This object can be used as a procedure to customize our button. For instance, setting the border of this button to 5 pixels wide and its background to gray would be done in Tcl with

```
.hello configure -border 5 -background gray
```

In STk this would be expressed as

```
(.hello 'configure '-border 5 '-background "gray")
```

Since keyword colon is replaced by a dash when a `Tk-command` is called, this expression could also have been written as:

```
(.hello 'configure :border 5 :background "gray")
```

2 Associating Callbacks to Tk-commands

Starting with version 3.0, STk callbacks are Scheme closures[1]. Apart scroll commands, callbacks are Schemes procedures without parameter. Suppose for example, that we want to associate a command with the previous `.hello` button. In Tcl, such a command can be expressed as

```
.hello configure -command {puts stdout "Hello, world"; destroy .}
```

In STk, we can write

```
(.hello 'configure :command (lambda ()
                                  (display "Hello, world\n")
                                  (destroy *root*)))
```

When the user will press the mouse left button, the closure associated to the `:command` option will be evaluated in the global environment. Evaluation of the given closure will display the message and call the `destroy` *Tk-command.*

Note: The root widget is denoted "." in Tcl. This convention is ambiguous with the dotted pair convention and the dot must be quoted to avoid problems. Since this problem arises so often, the variable `*root*` has been introduced in STk to denote the Tk main window.

Managing Widget Scrollbars

When using scrollbars, Tk library passes parameters to the widget associated to the scrollbar (and *vice versa*). Let us look at a text widget with an associated scrollbar. When the scrollbar is moved, the command of the associated widget is invoked to change its view. On the other side, when browsing the content of the text widget (with arrows for example), the scrollbar is updated by calling it's associated closure. Tk library passes position informations to scrolling closures. This informations are the parameters of the closure. Hereafter is an example implementing a text widget with a scrollbar (see the help pages for details):

[1] Old syntax for callbacks (i.e. strings) is always supported but its use is deprecated.

```
(text '.txt :yscrollcommand  (lambda l (apply .scroll 'set l)))
(scrollbar '.scroll :command (lambda l (apply .txt 'yview l)))

(pack .txt :side "left")
(pack .scroll :fill "y" :expand #t :side "left")
```

3 Tk bindings

Bindings are Scheme closures

The Tk `bind` command associates Scheme scripts with X events. Starting with version 3.0 those scripts must be Scheme closures[2]. Binding closures can have parameters. Those parameters are one char symbols (with the same conventions than the Tcl % char, see the `bind` help page for details). For instance, the following Tcl script

```
bind .w <ButtonPress-3> {puts "Press on widget %W at position %x %y"}
```

can be translated into

```
(bind .w "<ButtonPress-3>"
         (lambda (|W| x y)
            (format #t "Press on widget ~A at position ~A ~A\n" |W| x y)))
```

Note: Usage of verticals bars for the `W` symbol is necessary here because the Tk toolkit is case sensitive (*e.g.* `W` in bindings is the path name of the window to which the event was reported, whereas `w` is the width field from the event.

Bindings are chained

In Tk4.0 and later, bindings are chained since it is possible for several bindings to match a given X event. If the bindings are associated with different tags, then each of the bindings will be executed, in order. By default, a class binding will be executed first, followed by a binding for the widget, a binding for its toplevel, and an `all` binding. The `bindtags` command may be used to change this order for a particular window or to associate additional binding tags with the window (see corresponding help page for details). If the result of closure in the bindings chain is the symbol `break`, the next closures of the chain are not executed. The example below illustrates this:

```
(pack (entry '.e))
(bind .e "<KeyPress>" (lambda (|A|)
                        (unless (string->number |A|) 'break)))
```

Bindings for the entry `.e` are executed before those for its class (i.e. `Entry`). This allows us to filter the characters which are effectively passed to the `.e` widget. The test in this binding closure breaks the chain of bindings if the typed character is not a digit. Otherwise, the following binding, the one for the `Entry` class, is executed and inserts the character typed (a digit). Consequently, the simple previous binding makes `.e` a controlled entry which only accepts integer numbers.

[2]Old syntax for bindings (i.e. strings) is no more supported. Old bindings scripts must hence be rewritten.

Appendix B

Differences with R4RS

This appendix summarizes the main differences between the STk Scheme implementation and the language described in R^4RS.

1 Symbols

STk symbol syntax has been augmented to allow case significant symbols. This extension is discussed in 6.4.

STk also defines some symbols in the global environment which are described below:

- `*debug*`. Setting `*debug*` to `#t` prevents macro inlining and expression recoding (see 6.25).
- `*gc-verbose*`. If `*gc-verbose*` is `#t`, a message will be printed before and after each run of garbage collector. The message is printed on the standard error stream.
- `*load-verbose*`. If `*load-verbose*` is `#t`, the absolute path name of each loaded file is printed before its effective reading. File names are printed on the standard error stream.
- `*load-path*` must contain a list of strings. Each string is taken as a directory path name in which a file will be searched for loading. This variable can be set automatically from the `STK_LOAD_PATH` shell variable. See `stk(1)` for more details.
- `*load-suffixes*` must contain a list of strings. When the system try to load a file in a given directory (according to `*load-path*` value), it will first try to load it without suffix. If this file does not exist, the system will sequentially try to find the file by appending each suffix of this list. A typical value for this variable may be `("stk" "stklos" "scm" "so")`.
- `*argc*` contains the number of arguments (0 if none), not including interpreter options. See `stk(1)` for more details.
- `*argv*` contains a Scheme list whose elements are the arguments (not including the interpreter options), in order, or an empty list if there are no arguments. See `stk(1)` for more details.

- `*program-name*` contains the file name specified with the `-file` option, if present. Otherwise, it contains the name through which the interpreter was invoked. See `stk(1)` for more details.

- `*print-banner*`. If `*print-banner*` is `#f`, the usual copyright message is not displayed when the interpreter is started.

- `*stk-library*` contains the path name of the installation directory of the STk library. This variable can be set automatically from the `STK_LIBRARY` shell variable. See `stk(1)` for more details.

The following symbols are defined only when Tk is loaded:

- `*root*` designates the Tk main window (see A-2). This variable is not set if the Tk toolkit is not initialized.

- `*help-path*` must contain a list of strings. Each string is taken as a directory path name in which documentation files are searched. This variable can be set automatically from the `STK_HELP_PATH` shell variable. See `stk(1)` for more details.

- `*image-path*` must contain a list of strings. Each string is taken as a directory path name in which images are searched by the function `make-image`. This variable can be set automatically from the `STK_IMAGE_PATH` shell variable. See `stk(1)` and `make-image(n)` for more details.

- `*root*` designates the Tk main window (see A-2). This variable is not set if the Tk toolkit is not initialized.

- `*start-withdrawn*`. If *start-withdrawn* is not false, the *root* window is not mapped on screen until its first sub-window is packed or some action is asked to the window manager for it.

- `*tk-version*` is a string which contains the version number of the Tk toolkit used by STk.

- `*tk-patch-level*` is a string which contains the version and patch level of the Tk toolkit used by STk.

Furthermore, STk also defines the following procedures in the global environment:

- `report-error`. This procedure is called by the error system to display the message error. This procedure is described in `report-error(n)`

- `repl-display-prompt`. This procedure is called when the system is run interactively before reading a *sexpr* to evaluate to display a prompt. This procedure is described in `repl-display-prompt(n)`.

- `repl-display-result`. This procedure is called when the system is run interactively after the evaluation of a *sexpr* to write the result. This procedure is described in `repl-display-result(n)`.

2 Types

STk implements all the types defined as mandatory in R^4RS. However, complex numbers and rational numbers (which are defined but not required in R^4RS) are not implemented. The lack of these types implies that some functions of R^4RS are not defined.
Some types which are not defined in R^4RS are implemented in STk. Those types are listed below:

- input string port type (6.10)
- output string port type (6.10)
- keyword type (6.11)
- Tk command type (6.12)
- environment type (6.14)
- macro type (6.15)
- address type (6.17)
- hash table type (6.19)
- Regular expression type (6.20)
- process type (6.22)
- socket type (6.23)

3 Procedures

The following procedures are required by R^4RS and are not implemented in the STk interpreter.

- transcript-off
- transcript-on

`Transcript-off` and `transcript-on` can be simulated with various Unix tools such as `script` or `fep`.

The following procedures are not implemented in the STk interpreter whereas they are defined in R^4RS (but not required). They are all related to complex or rational numbers.

- numerator
- denominator
- rationalize
- make-rectangular
- make-polar

- real-part
- imag-part
- magnitude
- angle

Appendix C

An introduction to STklos

1 Introduction

STklos is the object oriented layer of STk. Its implementation is derived from version 1.3 of the Gregor Kickzales Tiny Clos package [12]. However, it has been extended to be as close as possible to CLOS, the Common Lisp Object System[6]. Some features of STklos are also issued from Dylan[13] or SOS[14].
Briefly stated, the STklos extension gives the user a full object oriented system with meta-classes, multiple inheritance, generic functions and multi-methods. Furthermore, the whole implementation relies on a true meta object protocol, in the spirit of the one defined for CLOS[15]. This model has also been used to embody the predefined Tk widgets in a hierarchy of STklos classes. This set of classes permits to simplify the core Tk usage by providing homogeneous accesses to widget options and by hiding the low level details of Tk widgets, such as naming conventions. Furthermore, as expected, using of objects facilitates code reuse and definition of new widgets classes.
The purpose of this appendix is to introduce briefly the STklos package and in no case will it replace the STklos reference manual (which needs to be urgently written now ...). In particular, methods relative to the meta object protocol and access to the Tk toolkit will not be described here.

2 Class definition and instantiation

2.1 Class definition

A new class is defined with the `define-class` macro. The syntax of `define-class` is close to CLOS `defclass`:

(`define-class` *class* (⟨superclass$_1$⟩ ⟨superclass$_2$⟩...)
  (⟨slot description$_1$⟩ ⟨slot description$_2$⟩...)
  ⟨metaclass option⟩)

The ⟨metaclass option⟩ will not be discussed in this appendix. The ⟨superclass⟩es list specifies the super classes of *class* (see 3 for more 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 3.3.

As an exemple, consider now that we have to define a complex number. This can be done with the following class definition:

```
(define-class  <complex> (<number>)
   (r i))
```

This binds the symbol `<complex>` to a new class whose instances contain two slots. These slots are called `r` an `i` and we suppose here that they contain respectively the real part and the imaginary part of a complex number. Note that this class inherits from `<number>` which is a pre-defined class (`<number>` is the super class of the `<real>` and `<integer>` pre-defined classes).[1].

3 Inheritance

3.1 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`, `F` use multiple inheritance: each class inherits from two previously defined classes. Those class definitions define a hierarchy which is shown in Figure 1. 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)
          ⟹ (d a b c f)
```

Note: The order of slots is not significant.

[1]With this definition, a `<real>` is not a `<complex>` since `<real>` inherits from `<number>` rather than `<complex>`. In practice, inheritance could be modified *a posteriori*, if needed. However, this necessitates some knowledge of the meta object protocol and it will not be shown in this document

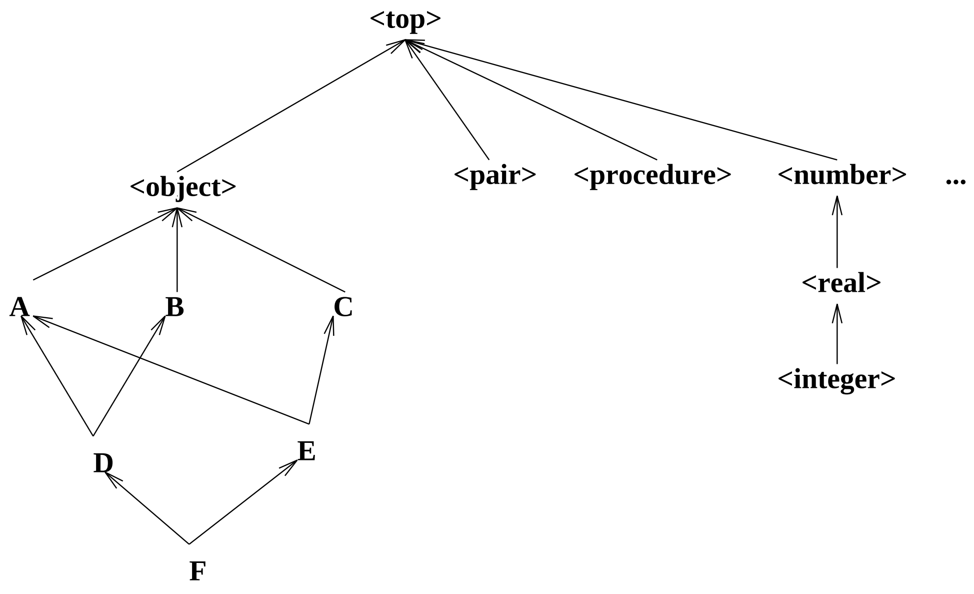

Figure C.1: A class hierarchy

3.2 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 <complex>))
```

will create a new `<complex>` object and will bind it to the `c` Scheme variable.
Accessing the slots of the new complex number can be done with the `slot-ref` and the `slot-set!` primitives. `Slot-set!` primitive permits to set the value of an object slot and `slot-ref` permits to get its value.

```
(slot-set! c 'r 10)
(slot-set! c 'i 3)
(slot-ref c 'r)
          ⟹ 10
(slot-ref c 'i)
          ⟹ 3
```

Using the `describe` generic 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)
```

will print the following informations on the standard output:

```
#[<complex> 122398] is an instance of class <complex>
Slots are:
     r = 10
     i = 3
```

3.3 Slot description

When specifying a slot, a set of options can be given to the system. Each option is specified with a keyword. The list of authorised keywords is given below:

- `:initform` permits to supply a default value for the slot. This default value is obtained by evaluating the form given after the `:initform` in the global environment.
- `:init-keyword` permits to specify the keyword for initializing a slot. The init-keyword may be provided during instance creation (i.e. in the make optional parameter list). Specifying such a keyword during instance initialization will supersede the default slot initialization possibly given with `:initform`.
- `:getter` permits to supply the name for the slot getter. The name binding is done in the global environment.
- `:setter` permits to supply the name for the slot setter. The name binding is done in the global environment.
- `:accessor` permits to supply the name for the slot accessor. The name binding is done in the global environment. An accessor permits to get and set the value of a slot. Setting the value of a slot is done with the extended version of `set!`.
- `:allocation` permits to specify how storage for the slot is allocated. Three kinds of allocation are provided. They are described below:
  - `:instance` indicates that each instance gets its own storage for the slot. This is the default.
  - `:class` indicates that there is one storage location used by all the direct and indirect instances of the class. This permits to define a kind of global variable which can be accessed only by (in)direct instances of the class which defines this slot.
  - `:virtual` indicates that no storage will be allocated for this slot. It is up to the user to define a getter and a setter function for this slot. Those functions must be defined with the `:slot-ref` and `:slot-set!` options. See the example below.

To illustrate slot description, we shall redefine the `<complex>` class seen before. A definition could be:

```
(define-class <complex> (<number>)
   ((r :initform 0 :getter get-r :setter set-r! :init-keyword :r)
    (i :initform 0 :getter get-i :setter set-i! :init-keyword :i)))
```

With this definition, the `r` and `i` slot are set to 0 by default. Value of a slot can also be specified by calling `make` with the `:r` and `:i` keywords. Furthermore, the generic functions `get-r` and `set-r!` (resp. `get-i` and `set-i!`) are automatically defined by the system to read and write the `r` (resp. `i`) slot.

```
(define c1 (make <complex> :r 1 :i 2))
(get-r c1)
          ⟹ 1
(set-r! c1 12)
(get-r c1)
          ⟹ 12
(define c2 (make <complex> :r 2))
(get-r c2)
          ⟹ 2
(get-i c2)
          ⟹ 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. So, another definition of the previous `<complex>` class, using the `:accessor` option, could be:

```
(define-class <complex> (<number>)
   ((r :initform 0 :accessor real-part :init-keyword :r)
    (i :initform 0 :accessor imag-part :init-keyword :i)))
```

Using this class definition, reading the real part of the `c` complex can be done with:

```
(real-part c)
```

and setting it to the value contained in the `new-value` variable can be done using the extended form of `set!`.

```
(set! (real-part c) new-value)
```

Suppose now that we have to manipulate complex numbers with rectangular coordinates as well as with polar coordinates. One solution could be to have a definition of complex numbers which uses one particular representation and some conversion functions to pass from one representation to the other. A better solution uses virtual slots. A complete definition of the `<complex>` class using virtual slots is given in Figure 2.

This class definition implements two real slots (`r` and `i`). Values of the `m` and `a` virtual slots are calculated from real slot values. Reading a virtual slot leads to the application of the function defined in the `:slot-ref` option. Writing such a slot leads to the application of the function defined in the `:slot-set!` option. For instance, the following expression

```
(slot-set! c 'a 3)
```

permits to set the angle of the `c` complex number. This expression conducts, in fact, to the evaluation of the following expression

```
(define-class <complex> (<number>)
   (;; True slots use rectangular coordinates
    (r :initform 0 :accessor real-part :init-keyword :r)
    (i :initform 0 :accessor imag-part :init-keyword :i)
    ;; Virtual slots access do the conversion
    (m :accessor magnitude :init-keyword :magn
       :allocation :virtual
       :slot-ref (lambda (o)
                   (let ((r (slot-ref o 'r)) (i (slot-ref o 'i)))
                     (sqrt (+ (* r r) (* i i)))))
       :slot-set! (lambda (o m)
                    (let ((a (slot-ref o 'a)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a))))))
    (a :accessor angle :init-keyword :angle
       :allocation :virtual
       :slot-ref (lambda (o)
                   (atan (slot-ref o 'i) (slot-ref o 'r)))
       :slot-set! (lambda(o a)
                    (let ((m (slot-ref o 'm)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a))))))))
```

Figure C.2: *A* `<complex>` *number class definition using virtual slots*

```
((lambda o m)
    (let ((m (slot-ref o 'm)))
       (slot-set! o 'r (* m (cos a)))
       (slot-set! o 'i (* m (sin a))))
  c 3)
```

A more complete example is given below:

```
(define c (make <complex> :r 12 :i 20))
(real-part c)
          ⟹ 12
(angle c)
          ⟹ 1.03037682652431
(slot-set! c 'i 10)
(set! (real-part c) 1)
(describe c)
          ⟹
          #[<complex> 128bf8] is an instance of class <complex>
          Slots are:
               r = 1
               i = 10
               m = 10.0498756211209
               a = 1.47112767430373
```

Since initialization keywords have been defined for the four slots, we can now define the `make-rectangular` and `make-polar` standard Scheme primitives.

```
(define make-rectangular
   (lambda (x y) (make <complex> :r x :i y)))

(define make-polar
   (lambda (x y) (make <complex> :magn x :angle y)))
```

3.4 Class precedence list

A class may have more than one superclass.[2] With single inheritance (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 :initform 1)))

(define-class Y ()
   ((x :initform 2)))

(define-class Z (X Y)
   (...))
```

In this case, the `Z` class is more specific than the `X` or `Y` class for instances of `Z`. However, the `:initform` specified in `X` and `Y` leads to a problem: which one overrides the other? The rule in STklos, as in CLOS, is that the superclasses listed earlier are more specific than those listed later. So:

Rule 2: For a given class, superclasses listed earlier are more specific than those listed later.

These rules are used to compute a linear order for a class and all its superclasses, from most specific to least specific. This order is called the "class precedence list" of the class. Given these two rules, we can claim that the initial form for the `x` slot of previous example is 1 since the class `X` is placed before `Y` in class precedence list of `Z`.
This two rules are not always enough to determine a unique order, however, but they give an idea of how things work. STklos algorithm for calculating the precedence list is a little simpler than the CLOS one described in [15] for breaking ties. Consequently the calculated class precedence list could be different. Taking the `F` class shown in Figure 1, the STklos calculated class precedence list is

```
(f d e a b c <object> <top>)
```

whereas it would be the following list with a CLOS-like algorithm:

```
(f d e a c b <object> <top>)
```

[2] This section is an adaptation of Jeff Dalton's (J.Dalton@ed.ac.uk) Brief introduction to CLOS)

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 B)
        ⟹ (#[<class> 12a248] #[<class> 1074e8] #[<class> 107498])
```

However, this result is not too much readable; using the function class-name yields a clearer result:

```
(map class-name (class-precedence-list B))
        ⟹ (b <object> <top>)
```

4 Generic functions

4.1 Generic functions and methods

Neither STklos nor CLOS use the message mechanism for methods as most Object Oriented language do. Instead, they use the notion of generic function. A generic function can be seen as a methods "tanker". When the evaluator requestd the application of a generic function, all the 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 funtion must be "called" the system will

1. search among all the generic function those which are applicable
2. sort the list of applicable methods in the "most specific" order
3. call the most specific method of this list (i.e. the first method of the sorted methods list).

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. Note that `define-method` automatically defines the generic function if it has not been defined before. Consequently, most of the time, the `define-generic` needs not be used.
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 will permit 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 to (directly or indirectly) to be applicable. If no speciliazer 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 #3 'a)
          ⟹ real
(M #t #f)
          ⟹ top
(M 1 2 3)
          ⟹ error (since no method exists for 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)
```

In this case,

```
(M 1 2)
          ⟹ integer-integer
(M 1 1.0)
          ⟹ integer-real
(M 1 #t)
          ⟹ integer
(M 'a 1)
          ⟹ 'top-number
```

4.2 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 4.1). This method may call 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,

```
(Test 1)
          ⟹ (integer number top)
(Test 1.0)
          ⟹ (number top)
(Test #t)
          ⟹ (top)
```

4.3 Example

In this section we shall continue to define operations on the `<complex>` class defined in Figure 2. Suppose that we want to use it to implement complex numbers completely. For instance a definition for the addition of two complexes could be

```
(define-method new-+ ((a <complex>) (b <complex>))
  (make-rectangular (+ (real-part a) (real-part b))
                    (+ (imag-part a) (imag-part b))))
```

To be sure that the `+` used in the method `new-+` is the standard addition we can do:

```
(define-generic new-+)

(let ((+ +))
  (define-method new-+ ((a <complex>) (b <complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                      (+ (imag-part a) (imag-part b)))))
```

The `define-generic` ensures here that `new-+` will be defined in the global environment. Once this is done, we can add methods to the generic function `new-+` which make a closure on the `+` symbol. A complete writing of the `new-+` methods is shown in Figure 3.

We use here the fact that generic function are not obliged to have the same number of parameters, contrarily to CLOS. The four first methods implement the dyadic addition. The fifth method says that the addition of a single element is this element itself. The sixth method says that using the addition with no parameter always return 0. The last method takes an arbitrary number of parameters[3]. This method acts as a kind of `reduce`: it calls the dyadic addition on the *car* of the list and on the result of applying it on its rest. To finish, the `set!` permits to redefine the `+ symbol` to our extended addition.

To terminate our implementation (integration?) of complex numbers, we can redefine standard Scheme predicates in the following manner:

```
(define-method complex? (c <complex>) #t)
(define-method complex? (c)           #f)

(define-method number? (n <number>) #t)
(define-method number? (n)          #f)
```

[3]The third parameter of a define-method is a parameter list which follow the conventions used for lambda expressions. In particular it can use the dot notation or a symbol to denote an arbitrary number of parameters

```
(define-generic new-+)

(let ((+ +))

  (define-method new-+ ((a <real>) (b <real>)) (+ a b))

  (define-method new-+ ((a <real>) (b <complex>))
    (make-rectangular (+ a (real-part b)) (imag-part b)))

  (define-method new-+ ((a <complex>) (b <real>))
    (make-rectangular (+ (real-part a) b) (imag-part a)))

  (define-method new-+ ((a <complex>) (b <complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                      (+ (imag-part a) (imag-part b))))

  (define-method new-+ ((a <number>))  a)

  (define-method new-+ () 0)

  (define-method new-+ args  (new-+ (car args) (apply new-+ (cdr args)))))

(set! + new-+)
```

Figure C.3: *Extending* `+` *for dealing with complex numbers*

...
...

Standard primitives in which complex numbers are involved could also be redefined in the same manner.
This ends this brief presentation of the STklos extension.

Appendix D

Modules: Examples

This appendix shows some usages of the STk modules. Most of the examples which are exhibited here are derived from the Tung and Dybvig paper [5].

Interactive Redefinition

Consider first the definitions,

```
(define-module A
  (export square)
  (define square
    (lambda (x) (+ x x))))

(define-module B
  (import A)
  (define distance
    (lambda (x y)
      (sqrt (+ (square x) (square y))))))
```

Obviously, the `square` function exported from `A` is incorrect, as we can see in its usage below:

```
(with-module B (round (distance 3 4)))
        ⟹ 4.0
```

The function can be redefined (*corrected*) by the following expression:

```
(with-module A
  (set! square
        (lambda (x) (* x x))))
```

And now,

```
  (with-module B (round (distance 3 4)))
        ⟹ 5
```

which is correct.

Lexical principle

This example reuses the modules `A` and `B` of previous section and adds a `Compare` module that exports the `less-than-4?` predicates, which states if the distance from a point to the origin is less than 4.

```
(define-module A
  (export square)
  (define square (lambda (x) (* x x))))

(define-module B
  (import A)
  (export distance)

  (define distance
    (lambda (x y) (sqrt (+ (square x) (square y))))))

(define-module Compare
  (import B)
  (define less-than-4? (lambda (x y) (< (distance x y) 4)))
  (define square       (lambda (x)   (+ x x))))
```

Consider now the call,

```
(with-module compare (less-than-4? 3 4))
          ⟹ #f
```

The call to `distance` done from `less-than-4?` indirectly calls the `square` procedure of module `A` rather than the one defined locally in module `Compare`.

Mutually Referential Modules

This example uses two mutually referential modules taht import and export to each other to implement mutually recursive even? and odd? procedures

```
(define-module Odd)   ;; Forward declaration

(define-module Even
  (import Odd)
  (export even?)
  (define even? (lambda (x) (if (zero? x) #t (odd? (- x 1))))))

(define-module Odd
  (import Even)
  (export odd?)
  (define odd? (lambda (x) (if (zero? x) #f (even? (- x 1))))))
```

Hereafter are some usages of theses procedures:

```
(with-module Odd (odd? 3))
          ⟹ #t
(with-module Odd (odd? 10))
          ⟹ #f

(with-module Even (even? 3))
          ⟹ #f
(with-module Even (even? 10))
          ⟹ #t
```

Appendix E

Changes

Introduction

This appendix lists the main differences[1] among the various recent versions of STk. Differences with older versions as well as implementation changes are described in the CHANGES file located in the main directory of the STk distribution.

Release 3.99.3

Release date: 09/30/98 Mains changes/modifications since 3.99.2 are:

- Tk version is 8.0.3
- Base64 Encoding/Decoding extension
- Locale extension to treat strings and character using locale information

Release 3.99.2

Release date: 04/27/98 Mainly a bugs correcting release.
New function: `write*` which handle circular structures. `Format` accepts now the special tag "`~W`" for circular structures writing.

Release 3.99.1

Release date: 04/27/98 Mainly a bugs correcting release

Release 3.99.0

Release date: 04/10/98
Changes can be classified in three categories:

- About Scheme
  - A module system has been added

[1]Only the differences which affect the language or new ports are reported here. In particular, internal changes, packages written in Scheme, STklos or performance enhancements are not discussed here.

- Integration of the Bigloo `match-case` and `match-lambda` primitives. Furthermore, the file *bigloo.stk* provides some compatibility between STk and bigloo modules.
- A simple Foreign Function Interface has been added.
- integrates the R^5RS `values` and `call-with-values`
- multi-line comments have been added.
- new file primitives: `remove-file`, `rename-file` and `temporary-file-name`.
- new list primitives: `append!`, `last-pair`, `remq`, `remv` and `remove`.
- `load`, `try-load` and `autoload?` can nw be called with a module as second parameter. If this second parameter is present, the loading is done in the environment of the given module.

• About Tk

- Integration of the Tk8.0 toolkit
- Buttons, Checkbuttons and Radiobuttons can use a `:variable` and `:textvariable` in a given environment. This environment is given with the new `:environment` option.

• About STklos

- The MOP of STklos is now very similar to the CLOS's MOP. In particular generic function has been added for controlling slot accesses, as well as numerous introspection functions.
- When a class is redefined, the instances and methods which uses it are redefined accordingly, as in CLOS (i.e. if anew slot is added in a class, all its – direct or indirect – instances will have the new slot added dynamically.

Release 3.1.1

Release date: 09/26/96
This release is a bug correction release. It corrects a lot of bugs. A lot of these bugs prevent to install it on some architectures.

Release 3.1

Release date: 07/24/96

- Version of Tk is now at Tk4.1 level.
- STk has been ported on Windows 95 and Windows NT.
- Ports can have a handler which is executed when port becomes readable or writable (see primitives `when-port-readable` and `when-port-writable`.
- Sockets in server mode allow multiple concurrent connection.
- STklos: Two new methods: `object-eqv?` and `object-equal?` which are called when applying `eqv?` or `equal?` to instances.
- New primitive:`setenv!`

Release 3.0

Release date: 01/22/96

- Version of Tk is at Tk4.0p2 level.
- Closures are fully supported by Tk. That means that a callback can be now a Scheme closure with its environment. GC problems with closures and usage of the dirty `address-of` are definitively gone.
- Strings can contain null charters (printing of strings is more friendly in write mode).
- Signals can be redirected to Scheme closures. The end of a GC is seen as a signal.
- Traces on variables are changed (and re-work now): the associated trace must be a thunk.
- New options for some widgets to be more friendly with Scheme world
- STklos: if a method M is defined and if it is already bound to a procedure, the old procedure is called when no method is applicable.

```
(define-method car ((x <integer>)) (- x 1))
(car 10)
          ⟹ 9
(car (cons 'a 'b))
          ⟹ a
```

- Small change in the STklos hierarchy. `<widget>` is now a subclass of `<procedure>` and its meta class is `<procedure-metaclass>`.

Appendix F

Miscellaneous Informations

1 Introduction

This appendix lists a number of things which cannot go elsewhere in this document. The only link between the items listed her is that they should ease your life when using STK.

2 About STK

2.1 Latest release

STK distribution is available on various sites. The original distribution site is `kaolin.unice.fr` `(134.59.132.7)`. Files are available through anonymous ftp and are located in the `/pub/STk` directory. Distribution file names have the form `STk-x.y.z.tar.gz`, where `x` and `y` represent the version the release and sub-release numbers of the package.

2.2 Sharing Code

If you have written code that you want to share with the (small) STK community, you can deposit it in the directory `/pub/STk/Incoming` of `kaolin.unice.fr`. Mail me a small note when you deposit a file in this directory so I can put in in its definitive place (`/pub/STk/Contrib` directory contains the contributed code).

2.3 STK Mailing list

There is a mailing list for STK located on `kaolin.unice.fr`. The intent of this mailing list is to permit to STK users to share experiences, expose problems, submit ideas and ... everything which you find interesting (and which is related to STK).
To subscribe to the mailing list, simply send a message with the word `subscribe` in the `Subject:` field of you mail. Mail must be sent to the following address: `stk-request@kaolin.unice.fr`
To unsubscribe from the mailing list, send a mail at previous email address with the word `unsubscribe` in the `Subject:` field.
For more information on the mailing list management send a message with the word `help` in the `Subject:` field of your mail. In particular, it is possible to find all the messages which have already been sent on the STK mailing list.
Subscription/un-subscription/information requests are processed automatically without human intervention. If you something goes wrong, send a mail to `eg@unice.fr`.
Once you have properly subscribe to the mailing list,

- you can send your messages about STK to `stk@kaolin.unice.fr`,

- you will receive all the messages of the mailing list to the email address you used when you subscribed to the list.

2.4 STk FAQ

Marc Furrer has set up a FAQ for STk. This FAQ is regularly posted on the STk mailing list. It can also be accessed through `http://ltiwww.epfl.ch/ furrer/STk/FAQ.html`. ASCII version of the FAQ is available from `http://ltiwww.epfl.ch/ furrer/STk/FAQ.txt`.

2.5 Reporting a bug

When you find a bug in STk, please send its description to the following address `stk-bugs@kaolin.unice.fr`. Don't forget to indicate the version you use and the architecture the system is compiled on. STk version and architecture can be found by using the `version` and `machine-type` Scheme primitives. If possible, try to find a small program which exhibit the bug.

3 STk and Emacs

The Emacs family editors can be customized to ease viewing and editing programs of a particular sort. Hints given below enable a fine "integration" of STk in Emacs.

Automatic scheme-mode setting

Emacs mode can be chosen automatically on the file's name. To edit file ended by `.stk` or `.stklos` in Scheme mode, you have to set the Elisp variable `auto-mode-alist` to control the correspondence between those suffixes and the scheme mode. The simpler way to set this variable consists to add the following lines in your `.emacs` startup file.

```
;; Add the '.stk' and '.stklos' suffix in the auto-mode-alist Emacs
;; variable. Setting this variable permits to automagically place the
;; buffer in scheme-mode.
(setq auto-mode-alist
      (append '(("\\.scm$"      .       scheme-mode)
                ("\\.stk$"      .       scheme-mode)
                ("\\.stklos$"   .       scheme-mode))
              auto-mode-alist))
```

Using Emacs and *CMU Scheme*

CMU Scheme package package permits to run the STk interpreter in an Emacs window. Once the package is loaded, you can send text to the inferior STk interpreter from other buffers containing Scheme source. The *CMU Scheme* package is distributed with Emacs (both FSF-Emacs and Xemacs) and you should have it if you are running this editor.
To use the *CMU Scheme* package with STk, place the following lines in your `.emacs` startup file.

```
;; Use cmu-scheme rather than xscheme which is launched by default
;; whence running 'run-scheme' (xscheme is wired with CScheme)
(autoload 'run-scheme "cmuscheme" "Run an inferior Scheme" t)
(setq scheme-program-name "stk")
(setq inferior-scheme-mode-hook '(lambda() (split-window)))
```

After having entered those lines in your `.emacs` file, you can simply run the STk interpreter by typing

```
M-x run-scheme
```

Read the *CMU Scheme* documentation (or use the describe-mode Elisp command) for a complete description of this package.

Using Emacs and the *Ilisp* package

Ilisp is another scheme package which allows to run the STk interpreter in an Emacs window. This is a rich package with a lot of nice features. *Ilisp* comes pre-installed with Xemacs; it has to be installed with FSF Emacs (the last version of *Ilisp* can be ftp'ed anonymously from `ftp.cs.cmu.edu` (128.2.206.173) in the `/user/ai/lang/lisp/util/emacs/ilisp` directory).
To use the *Ilisp* package with STk, place the following lines in your `.emacs` startup file.

```
(autoload 'run-ilisp            "ilisp" "Select a new inferior LISP." t)
(autoload 'stk                  "ilisp" "Run stk in ILISP." t)
(add-hook 'ilisp-load-hook
          '(lambda ()
             (require 'completer)

             ;; Define STk dialect characteristics
             (defdialect stk "STk Scheme"
               scheme
               (setq comint-prompt-regexp "^STk> ")
               (setq ilisp-program "stk -interactive")
               (setq comint-ptyp t)
               (setq comint-always-scroll t)
               (setq ilisp-last-command "*"))))
```

After having entered those lines in your `.emacs` file, you can simply run the STk interpreter by typing

```
M-x stk
```

The Ilisp package comes with a rich documentation which describe how to customize the package.

Other packages

Another way to use STk and Emacs consists to use a special purpose STk mode. You can find two such modes in the `/pub/Contrib` directory of `kaolin.unice.fr`.

3.1 Using the SLIB package with STk

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 STk `require` and `provide` are overloaded such as if their parameter is a string this is the old STk procedure which is called, and if their parameter is a symbol, this is the *SLIB* one which is called.

4 Getting information about Scheme

4.1 The R^4RS document

R^4RS is the document which fully describe the Scheme Programming Language, it can be found in the Scheme repository (see **??**) in the directory:

ftp.cs.indiana.edu:/pub/scheme-repository/doc

Aubrey Jaffer has also translated this document in HTML. A version of this document is available at

file://swiss-ftp.ai.mit.edu/pub/scm/HTML/r4rs_toc.html

4.2 The Scheme Repository

The main site where you can find (many) informations about Scheme is located in the University of Indiana. The Scheme repository is maintained by David Eby. The repository currently consists of the following areas:

- Lots of scheme code meant for benchmarking, library/support, research, education, and fun.
- On-line documents: Machine readable standards documents, standards proposals, various Scheme-related tech reports, conference papers, mail archives, etc.
- Most of the publicly distributable Scheme Implementations.
- Material designed primarily for instruction.
- Freely-distributable promotional or demonstration material for Scheme-related products.
- Utilities (e.g., Schemeweb, SLaTeX).
- Extraneous stuff, extensions, etc.

You can access the Scheme repository with

- `ftp.cs.indiana.edu:/pub/scheme-repository`
- `http://www.cs.indiana.edu/scheme-repository/SRhome.html`

The Scheme Repository is mirrored in Europe:

- `ftp.inria.fr:/lang/Scheme`
- `faui80.informatik.uni-erlangen.de:/pub/scheme/yorku`
- `ftp.informatik.uni-muenchen.de:/pub/comp/programming/languages/scheme/scheme-repository`

4.3 Usenet newsgroup and other addresses

There is a usenet newsgroup about the Scheme Programming language: `comp.lang.scheme`.
Following addresses contains also material about the Scheme language

- `http://www.cs.cmu.edu:8001/Web/Groups/AI/html/faqs/lang/scheme/top.html` contains the Scheme FAQ.
- `http://www-swiss.ai.mit.edu/scheme-home.html` is the Scheme Home page at MIT
- `http://www.ai.mit.edu/projects/su/su.html` is the Scheme Underground web page

Bibliography

[1] William Clinger and Jonathan Rees (editors). *Revised*^{4} Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), 1991.

[2] John K. Ousterhout. An X11 toolkit based on the Tcl Language. In *USENIX Winter Conference*, pages 105–115, January 1991.

[3] John K. Ousterhout. Tcl: an embeddable command language. In *USENIX Winter Conference*, pages 183–192, January 1990.

[4] Erick Gallesio. Extending the STK interpreter. Technical report, I3S CNRS / Université de Nice - Sophia Antipolis, 1997.

[5] Sho-Huan Simon Tung and R. Kent Dybvig. Reliable interactive programming with modules. *LISP and Symbolic Computation*, 9:343–358, 1996.

[6] Guy L. Steele Jr. *Common Lisp: the Language, 2nd Edition*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1990.

[7] POSIX Committee. *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990.

[8] Manuel Serrano. *Bigloo User's Manual, v1.9b*, June 1997.

[9] C. Queinnec and J-M. Geffroy. Partial Evaluation Applied to Symbolic Pattern Matching with Intelligent Backtrack. In *et al* M. Billaud, editor, *Workshop in Static Analysis*, number 81–82 in Bigre, Bordeaux (France), September 1992.

[10] A. Wright and B.Duba. Pattern Matching for Scheme. Technical report, Department of Computer Science, Rice University, October 1993.

[11] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994.

[12] Gregor Kickzales. Tiny-clos. Source available on parcftp.xerox.com in directory /pub/mops, December 1992.

[13] Apple Computer. *Dylan: an Object Oriented Dynamic Language*. Apple, April 1992.

[14] Chris Hanson. The sos reference manual, version 1.5. *in-line documentation of the SOS package. Source available on martigny.ai.mit.edu in /archive/cph directory*, March 1993.

[15] Jim de Rivières Gregor Kickzales and Daniel G. Bobrow. *The Art of Meta Object Protocol*. MIT Press, 1991.

Index

! 39
* 15
argc 73
argv 73
debug 37; 62, 73
gc-verbose 73
help-path 74
image-path 74
load-path 73
load-suffixes 73
load-verbose 73
print-banner 74
program-name 74
root 70; 74
start-withdrawn 74
stk-library 74
tk-patch-level 74
tk-version 74
+ 15
- 15
/ 16
:accessor 80
:allocation 80
:class 80
:getter 80
:init-keyword 80
:initform 80
:instance 80
:setter 80
:slot-ref 80; 81
:slot-set! 80; 81
:virtual 80
< 15
<= 15
<object> 78
<top> 78; 85
= 15
> 15
>= 15

A

abs 16
accessor 80
acos 16
add-signal-handler! 41
address-of 40
address? 40
and 9
angle 16; 76
append 12
append! 12
apply 20
apropos 62
asin 16
assoc 13
assq 13
assv 13
atan 16
autoload 27
autoload? 27

B

basename 38
begin 10
bind 71
bindtags 71
boolean? 11
break 40; 71
button 30
bye 65

C

c 41
c-string->string 58
caar 12
cadr 12
call-with-current-continuation 20
call-with-input-file 22
call-with-input-string 22
call-with-output-file 22
call-with-output-string 22
call-with-values 94
call/cc 20; 21
canonical-path 38
car 12
case 9

catch ... 21
cdddar ... 12
cddddr ... 12
cdr ... 12
ceiling ... 16
char->integer ... 18
char-alphabetic? ... 17
char-ci<=? ... 17
char-ci<? ... 17
char-ci=? ... 17
char-ci>=? ... 17
char-ci>? ... 17
char-downcase ... 18
char-lower-case? ... 18
char-numeric? ... 18
char-ready? ... 24
char-upcase ... 18
char-upper-case? ... 18
char-whitespace? ... 18
char<=? ... 17
char<? ... 17
char=? ... 17
char>=? ... 17
char>? ... 17
char? ... 17
characters ... 17
chdir ... 39
circular structures ... 8; 24, 25
class ... 77
class-precedence-list ... 84
class-slots ... 78
close-input-port ... 24
close-output-port ... 24
close-port ... 27
closure? ... 20
cmu scheme ... 98
complex? ... 15
cond ... 9
cons ... 12
continuation ... 20; 21, 60
continuation? ... 21
copy-tree ... 13
cos ... 16
current-error-port ... 22
current-input-port ... 22; 23, 24
current-module ... 33
current-output-port ... 22; 23, 26
currentdiscretionary --outputdiscretionary --port ... 24

D

decompose-file-name ... 38
default slot value ... 80
define-class ... 77; 78
define-external ... 56; 55
define-generic ... 84
define-macro ... 37
define-method ... 84
define-module ... 31; 33, 34
delay ... 10; 21
denominator ... 16; 75
describe ... 79
detail ... 64
dirname ... 38
display ... 25
do ... 10
dotimes ... 11
dump ... 60
dynamic-wind ... 21

E

emacs editor ... 98
environment->list ... 35
environment? ... 35
eq ... 13
eq? ... 12; 42, 43
equal ... 13
equal? ... 12
eqv ... 13
eqv? ... 11; 12
error ... 60
eval ... 59
eval-string ... 59
even? ... 15
exact->inexact ... 17
exact? ... 15
exec ... 40
exit ... 64
exp ... 16
expand-file-name ... 37
expand-heap ... 61
export ... 32
export-all-symbols ... 33
export-symbol ... 33
expt ... 16
external-exists? ... 58

F

faq ... 98
file-exists? ... 38
file-is-directory? ... 38
file-is-executable? ... 38
file-is-readable? ... 38

file-is-regular? ... 38
file-is-writable? ... 38
find-module ... 32
floor ... 16
fluid-let ... 10
flush ... 26
for-each ... 20
force ... 20
format ... 25; 60

G

garbage collector ... 57
gc ... 60
gc-stats ... 61
gcd ... 16
gensym ... 14
get-internal-info ... 61
get-keyword ... 29
get-output-string ... 23; 25
get-signal-handlers ... 42
get-widget-data ... 31
getcwd ... 39
getenv ... 40
getpid ... 39
getter ... 80
glob ... 38; 39
global-environment ... 35

H

hash-table->list ... 45
hash-table-for-each ... 44
hash-table-get ... 43
hash-table-hash ... 43; 42
hash-table-map ... 44
hash-table-put! ... 43
hash-table-remove! ... 44
hash-table-stats ... 45
hash-table? ... 43
help, getting ... 74

I

if ... 9
ilisp package ... 99
imag-part ... 16; 76
import ... 32
inexact->exact ... 17
inexact? ... 15
initial environment ... 11
input-port? ... 22
input-string-port? ... 22
inspect ... 62
instance ... 79
integer->char ... 18
integer? ... 15
ip number ... 52

K

keyword ... 80
keyword->string ... 29
keyword? ... 29

L

label ... 30
lambda ... 9
last-pair ... 13
lcm ... 16
length ... 12
let ... 10
let* ... 10
letrec ... 10
list ... 12; 13
list* ... 13
list->string ... 19
list->vector ... 20
list-ref ... 13
list-tail ... 13
list? ... 12
load ... 26
log ... 16

M

machine-type ... 59
macro ... 36; 37
macro-body ... 37
macro-expand ... 36; 37
macro-expand-1 ... 36
macro-expansion ... 36
macro? ... 36
magnitude ... 16; 76
make ... 79
make-client-socket ... 52
make-hash-table ... 42
make-keyword ... 29
make-polar ... 16; 75, 82
make-rectangular ... 16; 75, 82
make-server-socket ... 53; 52
make-string ... 18
make-vector ... 20
map ... 20
match-case ... 48; 94
match-lambda ... 48; 94
max ... 15

member 13
memq 13
memv 13
menu 30
min 15
module-environment 35
module-exports 34
module-imports 34
module-name 34
module-symbols 34
module? 32
modulo 16

N

negative? 15
newline 25
next-method 85
not 11
null? 12
number->string 17
number? 15
numerator 16; 75

O

obj 13; 40
odd? 15
open-file 27
open-input-file 23
open-input-string 23
open-output-file 23
open-output-string 23
or 9
output-port? 22
output-string-port? 22

P

pair? 12
parent-environment 35
pattern matching 47
peek-char 24
pid 50
port->list 28
port->sexp-list 28
port->string 28
port->string-list 28
positive? 15
posix.1 40
primitive? 20
procedure-body 21
procedure-environment 35
procedure? 20
process-alive? 50
process-continue 51
process-error 51
process-exit-status 51
process-input 51
process-kill 51
process-list 51
process-output 51
process-pid 50
process-send-signal 51
process-stop 51
process-wait 51
process? 50
promise? 21
provide 27; 99
provided? 27

Q

quasiquote 10
quit 64
quote 9
quotient 16

R

r4rs 7; 99
random 59
rational? 15
rationalize 16; 75
read 24; 8
read-char 24
read-from-string 59
read-line 24
real-part 16; 76
real? 15
regexp-replace 47
regexp-replace-all 47
regexp? 46
regular expression 45
remainder 16
remove 13
remove-file 39
remq 13
remv 13
rename-file 39
repl-display-prompt 74
repl-display-result 74
report-error 74
require 27; 99
reverse 13
root window 70
round 16

run-process 50

S

scheme repository 100
select-module 34
set! 9; 80, 81
set-car! 12
set-cdr! 12
set-random-seed! 59
set-signal-handler! 41
set-widget-data! 31
setenv! 40
setter 80
sigabrt 40
sigalrm 40
sigbus 41
sigchld 40
sigcld 41
sigcont 40
sigfpe 40
sighup 40
sigill 40
sigint 40; 41
sigio 41
sigiot 40
sigkill 40; 41
siglost 41
sigpipe 40
sigpoll 41
sigprof 41
sigsegv 40
sigstop 40; 41
sigsys 41
sigterm 40
sigtrap 40
sigttin 40
sigttou 40
sigurg 41
sigusr1 40
sigwinch 41
sigxcpu 41
sigxfsz 41
sin 16
slib package 99
slot 77
slot-ref 79
slot-set! 79
socket-accept-connection 53
socket-down? 53
socket-dup 54
socket-host-address 52
socket-host-name 52
socket-input 52
socket-local-address 52
socket-output 52
socket-port-number 52
socket-shutdown 53
socket? 52
sort 61; 39
split-string 19
sqrt 16
string 18; 19
string->list 19
string->number 17
string->regexp 45; 46
string->symbol 14
string->widget 30
string-append 19
string-ci<=? 19
string-ci<? 19
string-ci=? 18
string-ci>=? 19
string-ci>? 19
string-copy 19
string-fill! 19
string-find? 19
string-index 19
string-length 18
string-lower 19
string-ref 18
string-set! 18
string-upper 19
string<=? 18
string<? 18
string=? 18
string>=? 18
string>? 18
string? 18
substring 19
symbol->string 14
symbol-bound? 36
symbol? 14
system 39

T

tan 16
temporary-file-name 39
the-environment 35
tilde expansion 37
time 62
tk toolkit 7; 30, 60
tk-command 30; 31, 70
tk-command? 30
toolkit 7; 30, 60

top level environment 11; 35, 80
trace-var 60
transcript-off 27; 28, 75
transcript-on 27; 28, 75
truncate 16
try-load 27

U

uncode 61
unless 9; 11
until 11
untrace-var 60

V

values 94
vector 20
vector->list 20
vector-copy 20
vector-fill! 20
vector-length 20
vector-ref 20
vector-resize 20
vector-set! 20
vector? 20
version 59
view 62; 64

W

when 9
when-port-readable 26
when-port-writable 26
when-socket-ready 54
while 11
widget 30
widget->string 30
widget-name 31
widget? 30
with-input-from-file 22
with-input-from-string 23
with-module 33
with-output-to-file 22
with-output-to-string 23
write 24
write* 24
write-char 25
writes* 25

X

x window system 7

Z

zero? 15