4.3g

Bigloo

A practical Scheme compiler (4.3g)
User manual for version 4.3g
December 2019

Manuel Serrano

Copyright © 1992-99, 2000-02 Manuel Serrano

This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public
License along with this program; if not, write to the Free
Software Foundation, Inc., 59 Temple Place - Suite 330, Boston,
MA 02111-1307, USA.

Acknowledgements

Bigloo has been developed at Inria-Rocquencourt by the Icsla team from 1991 to 1994, at the University of Montreal in 1995 and at Digital's Western Research laboratory in 1996, University of Geneva during 1997 and from the end of 1997 at the University of Nice.

I would like to express my gratitude to *Hans J. Boehm* for his Garbage Collector [BoehmWeiser88, Boehm91], *Jean-Marie Geffroy* for his pattern-matching compiler [QueinnecGeffroy92], *Dominique Boucher* for his Lalr grammar compiler, *William Clinger* for his syntax expansion implementation and *Dorai Sitaram* for his contribution with the `pregexp` package and its documentation. I also especially thank *Christian Queinnec* for all his useful remarks, help, suggestions and teaching.

Other people have helped me by providing useful remarks, bug fixes or code improvements. I thank all of them and especially *Luc Moreau*, *John Gerard Malecki*, *David Halls* and *David Gurr*.

I thank *Barrie Stott* for his help in making much of the documentation more idiomatic. Of course, any remaining errors are still mine.

This release of Bigloo may still contain bugs. If you notice any, please forgive me and send a mail message to the following address: `bigloo@sophia.inria.fr`.

This is Bigloo documentation version 4.3g, December 2019.

1 Overview of Bigloo

Bigloo is an implementation of an extended version of the Scheme programming language. Without its extensions Bigloo does not entirely conform to Scheme as defined in the Revised(5) Report on on the Algorithmic Language Scheme (henceforth R5RS) (see `r5rs.info`). The two reasons are:

- Bigloo produces C files. C code uses the C stack, so some programs can't be properly tail recursive. Nevertheless all simple tail recursions are compiled without stack consumption.
- Alternatively, Bigloo may produce JVM (Java Virtual Machine byte code) class files. These classes may use regular Java classes.
- Bigloo is a module compiler. It compiles modules into '.o', '.class', or '.obj' files that must be linked together to produce stand alone executable programs, JVM jar files, or .NET programs.

However, we designed Bigloo to be as close as possible to the R5RS. Hence, when Bigloo includes a feature that is extracted from Scheme and implemented as normal, this feature is only mentioned in this document and not fully described.

1.1 SRFI

The Bigloo version 4.3g supports the following SRFIs:

- `srfi-0` (conditional execution).
- `srfi-2` (AND-LET*: an AND with local bindings, a guarded LET* special form).
- `srfi-6` (Basic String Ports).
- `srfi-8` (Binding to multiple values).
- `srfi-9` (*Records* specification).
- `srfi-18` (Multithreading support).
- `srfi-22` (script interpreter invocation).
- `srfi-28` (Basic Format Strings).
- `srfi-30` (Multi-line comments).

1.2 Separate compilation

To allow and stimulate separate compilation, Bigloo compiles modules instead of entire programs. A module is composed of a module declaration and a module body, where a module body can be thought of as an incomplete Scheme program.

Bigloo strictly enforces variable bindings. That is, it is illegal in a body to refer to unbound variables.

In a module declaration, some variables can be declared to be immutable functions. For such variables, the compiler can then check if the number of arguments for some function calls are correct or not. When an arity mismatch is detected, Bigloo signals an error and aborts the compilation process.

1.3 C interface

The goal of the design of Bigloo is to allow the merging of high and low level programming. This means that Bigloo is designed to be fully connected to the already existing outside world of C.

This connection has two components: a function call interface and a data storage interface. Bigloo code is able to call C code and vice versa; Bigloo data storage is accessible from C and vice versa. There are no frontiers between the Bigloo and C worlds. See Chapter 26 [C Interface], page 235, for details.

1.4 Java interface

Since release 2.3, Bigloo is able to produce Java Virtual Machine byte codes in addition to C code. By producing class files, it is possible to connect Scheme code and Java code in the same spirit as the Scheme and C connection.

This connection has two components: a function call interface and a data storage interface. Bigloo code is able to call Java code and vice versa; Bigloo data storage is accessible from Java and vice versa. There are no frontiers between the Bigloo and Java worlds. See Chapter 27 [Java Interface], page 247, for extra details.

1.5 Object language

Since release 1.9, Bigloo has included an object system. This system belongs to the Clos [Bobrow et al. 88] object system family but whose design has been mainly inspired by C. Queinnec's Meroon [Queinnec93]. It is based on *ad-hoc* polymorphism (generic functions and methods), uses single inheritance and mono-dispatch, and provides the user with introspection facilities.

1.6 Threads

Since release 2.4d, Bigloo has included a thread library. Bigloo supports Fair threads that are cooperative threads run by a fair scheduler which gives them equal access to the processor. Fair threads can communicate using broadcast events and their semantics does not depends on the executing platform. Fine control over fair threads execution is possible allowing the programming of specific user-defined scheduling strategies.

1.7 SQL

Since release 2.7b, Bigloo includes a SQL binding. Namely, the C Bigloo runtime system can access the facilities offered by SQLite (`http://www.sqlite.org/`).

1.8 Type annotations

Type information, related to variable or function definitions, can be added to the source code. If no type information is provided, runtime checks will be introduced by the compiler to ensure normal execution, provided that the user has not used compilation flags to prevents this. If type information is added, the compiler statically type checks the program and refuses ones that prove to be incorrect.

1.9 Unicode support

Bigloo supports UCS-2 Character encoding and also provides conversion functions between UTF-8 and UCS-2. It still maintains traditional ISO-LATIN1 characters and strings.

1.10 DSSSL

Bigloo helps the DSSSL programmer by supporting keywords, named constants and keyword functions.

2 Modules

A modules is a compiler and interpreter entity. Modules have been first designed for the compiler that compiles modules and then, links them against libraries in order to produce executables. A module may be split into several files but a file cannot contain more than *one* module. A module is made of a module clause that is a list for which the `car` is the symbol `module` and followed by any Bigloo expression (that is definitions or expressions). The module clause names the module and defines the scope of the definitions. At last, the module clause is also the place where foreign bindings are defined and where classes are defined. Recent versions of Bigloo (since 2.7b) fully supports modules from the interpreter.

2.1 Program Structure

A Bigloo program is composed of one or more Bigloo modules where a module is defined by the following grammar:

```
<module>             ↦ <module-declaration> <module-body>
<module-declaration> ↦ the module declaration
<module-body>        ↦ the module body
```

A module is not related to a specific file and can be spread over several files if that is convenient. In particular, there is no relationship between module names and file names. The module declaration (see Section 2.2 [Module Declaration], page 7) must be the first expression in the first of the files containing the module; other expressions form the body of the module. The module body (see Chapter 3 [Core Language], page 17) contains global variables, function definitions and *top level* expressions (see Section 3.1.2 [Expressions], page 17).

2.2 Module declaration

The module declaration form is

`module` *name clause . . .* [bigloo syntax]

This form defines a module and must be the first in the file. The argument *name* is a symbol naming the module. If the same module name is used more than once, Bigloo signals an error. The runtime library is composed of modules that are read when a user module is compiled and hence, if a user module has the same name as one of the library modules, an error is signaled.

A simple module can be:

```
(module foo)

(display "this is a module")
```

The first line here is the complete module definition, the last line is the complete module body and together they form a complete Bigloo program. If these lines were stored in file `zz.scm`, invoking '`bigloo zz.scm`' would create the executable `a.out` which, when obeyed, would display '`this is a module`' on the terminal.

Note: Some special identifiers are reserved and can't be used to name modules. If such an identifier is used, the compiler will produce the message:

```
#(module t
#^
```

```
# *** ERROR:bigloo:TOP-LEVEL:Parse error
# Illegal module name -- (MODULE eval ...
```

The list of reserved identifiers may be enlarged for next release. For the current release that list is made of: `eval`, `foreign` and `t`.

Module *clause*s can be:

`main` *name* [bigloo module clause]

This clause defines the entry point for a stand alone application to be procedure *name* of arity one. Bigloo invokes this procedure at the beginning of execution providing the list, composed of the shell command line arguments, as its single argument.

```
(module foo
   (main start))

(define (start argv)
   (display argv)
   (newline))
```

Then if this program is compiled into `foo` and invoked using the command '`foo -t bar`', the list which is the argument for the main procedure `start` would be `("foo" "-t" "bar")`.

The special form `args-parse` helps main function argument parsing (see Chapter 13 [Command Line Parsing], page 153).

`include` *file-name ...* [bigloo module clause]

This is a list of *file-name*s to be included in the source file. Include files are not modules and may have a special syntax. Thus, besides containing Bigloo expressions, they can contain import and include clauses, which must be written in a single list whose first element is the keyword `directives`. Includes files can be used to include implementation-neutral Scheme expressions and definitions in a Bigloo module. Here is an example of an include file.

```
;; foo.sch
(define-struct point x y)
```

and the module that includes the `foo.sch` file:

```
;; foo.scm
(module foo
   (include "foo.sch"))

(print (point 1 2))
```

Include files, may contain module information. This is the role of the include `directives` clause here illustrated with the `bar.sch` example:

```
;; bar.sch
;; the directives
(directives (include "foobar.sch")
            (import  hux))

;; expressions
(define (gee x) (print x))
```

`import` *import ...* [bigloo module clause]

An *import* is a list of the form:

```
<import>      ↦ <iclause> ...
```

```
<iclause>     ↦ (<bind-name> ... <bind-name> <module-name> <file-name> ...)
                | (<bind-name> ... <bind-name> <module-name>)
                | <module-name>
                | (<module-name> <file-name> ...)
<bind-name>   ↦ <r5rs-ident>
                | <alias-name>
<alias-name>  ↦ (<r5rs-ident> <r5rs-ident>)
<module-name> ↦ <r5rs-ident>
<file-name>   ↦ <string>
```

The first alternative in *iclause* imports the variable named *bind-name* which is defined in the module *module-name*, located in the files *file-name* .... The second does the same but without specifying the name of the file where the module is located. The third and the fourth form import all the exported variables of the module *module-name*.

Note: The need for specifying in which files modules are located comes from the fact that there is no automatic mapping between module names and files names. Such a mapping can be defined in a "module access file" (see Section 2.6 [Module Access File], page 16) or in the import clause itself, as in the first and fourth alternatives in *iclause* above.

Here is an example of an import clause:

```
(module foo
   (import
      ;; import all bar exported bindings:
      bar
      ;; import the hux binding exported by
      ;; the module hux:
      (hux hux)
      ;; import the fun1, fun2 and fun3 bindings exported by
      ;; the module mod:
      (fun1 fun2 fun3 mod)
      ;; import the fun4 bindings that will be known in this module
      ;; under the alias name f
      ((f fun4) mod)
      ;; import all gee bindings. the gee module
      ;; is located in a file called gee.scm:
      (gee "gee.scm")))
```

`use` *use ...* [bigloo module clause]

`use` has the same meaning as `import` except that modules which are *used* are not initialized (see Section 2.3 [Module Initialization], page 13). Used modules are read before imported modules.

`with` *with ...* [bigloo module clause]

This clause specifies a list of modules which have to be initialized at runtime and is used to force the initialization of modules which are never imported but which are required by an application (see Section 26.4 [Embedded Bigloo applications], page 245).

`export` *export ...* [bigloo module clause]

In order to make a module's global bindings available to other modules, they have to be *exported*. Export clauses are in charge of this task and an *export* is a list of the form:

```
<export>  ↦ <eclause> ...
<eclause> ↦ <ident>
            | (inline <ident> <ident> ...)
            | (generic <ident> <ident> <ident> ...)
            | (<ident> <ident> ...)
            | <class>
            | (macro <ident> <ident> ...)
            | (expander <ident>)
            | (syntax <ident>)
```

The first form of *eclause* allows the variable *ident* be exported, the second allows the function *ident*, always regarded as immutable when exported this way, to be exported and the third exports an inline-procedure (see Section 2.5 [Inline Procedures], page 15) whose name is extracted from the first *ident* after the word *inline*. The last two are both connected with Bigloo's object system. The *generic* clause exports generic functions (see Section 9.3 [Generic functions], page 121) and *class* clause exports classes (see Section 9.1 [Class declaration], page 117).

Note: Only bindings defined in module `m` can be *exported* by `m` (i.e. bindings *imported* by `m` cannot be *exported* by `m`).

Type information, specified in any *ident* in an export clause, is used by Bigloo. Where no type information is given, a default generic type named `obj` is used.

Note: The last formal argument of a multiple arity function can not be typed because this argument is bound to be a `pair` or `null`. This union cannot be denoted by any type.

Here is an example of the module `foo` that exports bindings:

```
(module foo
   (export
      ;; export the bar mutable variable
      bar
      ;; export the hux function. this
      ;; function takes exactly two arguments
      (hux x y)
      ;; export the inline function gee
      ;; that takes at least one argument.
      (inline gee x . z)))
```

`static` *static ...* [bigloo module clause]

A `static` clause has exactly the same syntax as an export clause. However, bindings declared static are local to the module. Since the default scope of all bindings is static, `static` module clauses are useful only for program documentation.

`from` *from ...* [bigloo module clause]

`from` clauses have the syntax of `import` clauses. The allow the re-exportation of imported bindings. That is, any module can export any bindings imported via a *from* clause.

As an example, suppose we have module `bar`:

```
(module bar
   (export (fun)))

(define (fun) "bar")
```

Now, suppose we have a module `foo` that imports `bar`, by the means of a `from` clause. Module `foo` is able to re-export the `bar` binding of module `bar`:

```
(module foo
   (from (fun bar "bar.scm")))
```

A third module, let's name it `gee`, importing module `foo`, can see the binding for function `bar`:

```
(module gee
   (import (foo "foo.scm")))

(print (fun))
```

This feature is very useful when compiling modules exporting functions with type annotations. In particular, one may write:

```
(module foo
  (export (class c1 x)))
```

Then,

```
(module bar
  (import foo)
  (from foo)
  (export (fun::c1)))

(define (fun)
   (instantiate::c1 (x 10)))
```

And,

```
(module gee
   (import bar)
   (main main))

(define (main x)
   (let ((o (fun)))
      (print o)
      (print (c1? o))))
```

`load` *load ...* [bigloo module clause]

A *load* is a list of the form:

```
<load>    ↦ <lclause> ...
<lclause> ↦ (<module-name> <file-name>)
            | <module-name>
```

This clause forces Bigloo to load the module specified in the *lclause* in the environment used by the macro expansion mechanism. This means that the user's macros can use all the bindings of all the `load`ed modules but the `load`ed bindings remains unknown to the compiler.

If the module `foo` is defined by:

```
(module foo
   (export (foo x)))

(define (foo x)
   `(cons ,x ,x))
```

then,

```
(module gee
   (load (foo "foo.scm")))
```

```
(define-macro (gee x)
   `(cons ,(-fx x 1) ,(foo x)))

(gee 5)   ↦ (cons 4 (cons 5 5))
          ⇒ (4 5 . 5)
```

`eval` *eval...* [bigloo module clause]

This form allows interactions between compiled code and interpreted code. (See the Section Chapter 22 [Eval command line options], page 223, for a presentation of compilation flags that enable compilation tuning for `eval`.) Each *eval* has the following syntax:

```
<eval> ↦ (export-all)
         | (export-module)
         | (export-exports)
         | (export <bind-name>)
         | (export (@ <bind-name> <module-name>))
         | (import <bind-name>)
         | (class <bind-name>)
         | (library lib1 ...)
```

The first clause, `(export-all)`, exports all the variables bound in the module (i.e., the variables defined in the module and the imported variables). The second clause, `(export-module)`, exports all the module's variables (those declared static and exported) to the interpreter; the third exports all the exports (i.e. the ones present inside an `export` clause) variables to the interpreter; the fourth and fifth clause each export one variable to the interpreter. The last clause imports a variable from the interpreter and all such imported variables are immutable (i.e. they cannot be the first argument of a `set!` expression with the compiled code). Variables that are exported to the evaluators *must* be exported. If a variable is exported to the evaluators but not exported within an `export` clause, the compiler will produce an error message. The `library` clause makes the variables and functions of a library accessible from the interpreter.

```
(module foo
   (export (fib x))
   (eval (export fib)
         (import bar)))

(define (fib x) ...)
(print bar)
```

The clause `(class <bind-name>)` exports a class definition to the interpreter. This makes the class constructor, the class predicate and the slots access functions available from the interpreter. The form `(instantiate::`*class* `...)` and `(with-access::`*class* `...)` are also available from the interpreter.

`extern` *extern ...* [bigloo module clause]

Extern (aka foreign) clauses will be explained in the foreign interface (see Chapter 26 [C Interface], page 235).

`java` *java ...* [bigloo module clause]

Java clauses will be explained in the Java interface (see Chapter 27 [Java Interface], page 247).

`option` *option ...* [bigloo module clause]

This clause enables variables which affect compilation to be set from inside a module and since the expressions, *option ...*, are evaluated *when compiling*, no code is compiled for them. They are allowed to make side effects and to change the values of the global variables which describe how the compiler must compile. Usually they allow the control variables, which are described when Bigloo is invoked with the `-help2` option, to be set as in the following example:

```
(module examplar
   (option (set! *debug* 3)
           (set! *verbose* 2)))

(print 'dummy)
```

Whatever arguments are passed on the command line, Bigloo will compile this module in both verbose mode and debug mode.

`library` *library ...* [bigloo module clause]

This clause enables libraries (see Chapter 28 [Bigloo Libraries], page 255) when compiling and linking Bigloo modules. The expressions *library ...* are symbols naming the libraries to be used.

Here is an example of a module declaration which makes use of a library named `format`:

```
(module test
   (library format)
   (main    test-format)
   (import  (test2 "test2.scm")))
```

Using a library does not automatically binds its variables and functions to the interpreter. In order to make these available to the interpreter an explicit use of an eval `library` clause must be used.

`type` *type ...* [bigloo module clause]

This forms is used to define builtin Bigloo types. It is not recommended to use it in user programs. So, it is left undocumented.

2.3 Module initialization

Initializing a module means evaluating, at runtime, its top level forms (global bindings are top level forms).

When a module, *module1*, imports a module, *module2*, *module2* is initialized before *module1*. Modules are initialized only once, nothing being done if a module already met during initialization is met again. Library modules are initialized before user modules and imported modules are initialized in the same order as they appear in import clauses.

Here is a first example with two modules. First the module `foo`:

```
;; module foo
(module foo
   (main main)
   (import (bar "bar.scm")))

(define (main argv)
   (print "argv: " argv))
```

```
(print "foo")
```

Then the module `bar`

```
;; module bar
(module bar)

(print "bar")
```

These can be compiled into the executable *a.out* with:

```
$ bigloo -c foo.scm
$ bigloo -c bar.scm
$ bigloo foo.o bar.o
```

Execution of *a.out* produces:

```
$ a.out
   ⊣ bar
     foo
     argv: (a.out)
```

The explanation is:

- module `foo` contains the program entry point so this is where initialization begins.
- because `foo` imports module `bar`, `bar` must be initialized *before* `foo`. This explains why the word `bar` is printed before anything else.
- module initialization for `foo` is completed before `main` is called. This explains why word `foo` is printed before `main` is entered.

Let's consider another example with 3 modules:

```
;; module1
(module module1
   (main main)
   (import (module2 "module2.scm")))

(define (main argv)
   (print "argv: " argv))

(print "module1")
```

The second module:

```
;; module2
(module module2
   (import (module3 "module3.scm")))

(print "module2")
```

The third module:

```
;; module3
(module module3
   (import (module1 "module1.scm")))

(print "module3")
```

Compile with:

```
$ bigloo module1.scm -c
$ bigloo module2.scm -c
$ bigloo module3.scm -c
$ bigloo module1.o module2.o module3.o
```

Execution produces:

```
$ a.out
```

```
⊣ module3
  module2
  module1
  argv: (a.out)
```

The order of module initialization can be explicitly specified using `with` and `use` clauses.

2.4 Qualified notation

Global variables can be referenced using implicit notation or using *qualified* notation. Implicit notation is used when variables are referenced just by their name whereas qualified notation is used when variables are referenced by their name and the name of the module which defines them. Qualified notation has the following syntax:

```
(@ <bind-name> <module-name>)
```

and is useful when several imported modules export a variable with the same name. Using qualified notations instead of short notation only affects compilation.

When several variables are defined under the same identifier, the compiler uses the two following rules in order to decide which variable is selected by an implicit reference: 1) the variable defined in a module has a higher precedence than all imported variables, 2) imported variables have a higher precedence than library variables.

2.5 Inline procedures

Bigloo allows procedures called *inline* and which differ from normal ones only in the type of code planted. An inline procedure is a first class object which can be manipulated in the same way as any other procedure but when Bigloo sees a reference to one, rather than generating a C function call to the function, the body of the inline procedure is open-coded. The definition of an inline is given in the following way:

`define-inline` (*name args . . .*) *body* [bigloo syntax]
`define-inline` (*name args . . . . arg*) *body* [bigloo syntax]

Apart from the initial word, this form has the same syntax as that used by `define` for procedures. Inline procedures are exportable which means that the compiler scans imported files to find the bodies of all inline procedures. Here is a small example of a module which exports an inline and a module which imports it.

```
;; the exporter module
(module exporter
        (export (inline make-list . objs)))

(define-inline (make-list . objs) objs)

;; the importer module
(module importer
        (import exporter))

(print (make-list 1 2 3 4 5))
```

Because of the open-coding of the exporter procedure, the above print statement is equivalent to:

```
(print (let ((objs (list 1 2 3 4 5)))
          objs))
```

Any procedure can be an inline. Also any exported procedure can be an inline provided all global variables and functions it uses are also exported.

Note: Bigloo can decide to inline procedures declared with `define` but this can be achieved only with local procedures whereas procedures declared with the `define-inline` form are open-coded even through module importation.

Note: Procedures declared *inline* are macro expanded with the macro defined in the module where they are invoked. That is, if module `module1` declares an inline procedure `p` and module `module2` imports it, `p` may have two different macro-expansions: one for `module1` and one for `module2`.

2.6 Module access file

Bigloo is different from languages such as C where a module is defined by a file. For Bigloo, the module name is not necessarily the name of the file where the text of the module is written and modules can even be split across several files.

Since modules are defined independently of files, it is necessary to make a link between a module and its files and there are two ways of doing this. Choosing an import clause where the file-names are specified or creating a "module access file". Such a file must contain only one *list*, each element of the list being of the form:

(module-name "file-name" ... "file-name")

Use the '`-afile <file>`' option to specify the "module access file" when compiling. By default Bigloo checks if a file named `.afile` exists. If it exists it is loaded as a module access file.

See Chapter 31 [The Bigloo command line], page 275.

Note: The Bigloo distribution contains a tool, `bglafile`, that can automatically build a "module access file". See the manpage for `bglafile` for details.

2.7 Reading path

Imported, included or loaded files are sought first in the current directory and then in the directories, sequentially from start to end, of the list in the `*load-path*` variable. This variable, initially set to the empty list, can be reset by the '`-I`' option of the compiler.

3 Core Language

This chapter presents the Bigloo basics. It presents the elements that compose the body of a module (see Chapter 2 [Modules], page 7).

3.1 Syntax

The syntax of Bigloo is that of Scheme (a parenthesis based one) with two exceptions: type information and multi-line comments. Type information is supplied when identifiers are introduced (via `lambda`, `let`, `define`, ...) and those identifiers holding type information are referred to as typed identifiers.

They are defined by the following grammar:

```
<ident>        ↦ <r5rs-ident> | <typed-ident>
<typed-ident>  ↦ <r5rs-ident>::<r5rs-ident>
<r5rs-ident>   ↦ the standard Scheme identifiers
```

For details of the standard Scheme identifiers, see Section "Lexical structure" in *R5RS*.

Multi-lines comments (see `http://srfi.schemers.org/srfi-30/`) are defined as:

```
<ident>        ↦ <r5rs-ident> | <typed-ident>
<comment>      ↦ ;<all subsequent characters up to a line break>
               | #| <comment-text> (<comment> <comment-text>)* |#
<comment-text> ↦ <character sequence not containing #| or |#>
```

3.1.1 Comments

Comments and whitespaces are the same as in Section "r5rs.info" in `R5RS`.

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
        1        ;; Base case: return 1
        (* n (fact (- n 1))))))
```

In addition, Bigloo supports *s-expressions* comments. These are introduced with the `#;` syntax:

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    #;(if (< n 2) 1 (* #;n (fact (- n 1))))
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

3.1.2 Expressions

Bigloo expressions are the same as in Section "r5rs.info" in `R5RS`. Bigloo has more syntactic keywords than Scheme. The Bigloo syntactic keywords are:

```
=>                  do                  or
and                 else                quasiquote
begin               if                  quote
case                lambda              set!
```

`cond`	`let`	`unquote`
`unquote-splicing`	`define`	`let*`
`delay`	`letrec`	`module`
`labels`	`try`	`define-struct`
`unwind-protect`	`bind-exit`	`define-inline`
`regular-grammar`	`lalr-grammar`	`regular-search`
`define-expander`	`define-macro`	`match-case`
`match-lambda`	`pragma`	`failure`
`assert`	`define-generic`	`define-method`
`instantiate`	`duplicate`	`with-access`
`widen!`	`shrink!`	`multiple-value-bind`
`let-syntax`	`letrec-syntax`	`define-syntax`
`cond-expand`	`receive`	`args-parse`
`define-record-type`	`and-let*`	`letrec*`

All other non atomic Bigloo forms are evaluated as function calls or macro class.

<variable> [syntax]

`quote` *datum* [syntax]

'datum [syntax]

<constant> [syntax]

```
(define x 28)                    ⇒
x                                ⇒ 28
(quote a)                        ⇒ A
(quote #(a b c))                 ⇒ #(A B C)
(quote (+ 1 2))                  ⇒ (+ 1 2)
'a                               ⇒ A
'#(a b c)                        ⇒ #(A B C)
'()                              ⇒ ()
'(+ 1 2)                         ⇒ (+ 1 2)
'(quote a)                       ⇒ (QUOTE A)
'"abc"                           ⇒ "abc"
"abc"                            ⇒ "abc"
'145932                          ⇒ 145932
145932                           ⇒ 145932
'#t                              ⇒ #t
#t                               ⇒ #t
```

`operator` *operand ...* [syntax]

```
(+ 3 4)                          ⇒ 7
((if #f + *) 3 4)                ⇒ 12
((lambda (x) (+ 1 x)) 5)         ⇒ 6
```

`lambda` *formals body* [syntax]

```
(lambda (x) (+ x x))             ⇒ a procedure
((lambda (x) (+ x x)) 4)         ⇒ 8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)          ⇒ 3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                         ⇒ 10
```

```
((lambda x x) 3 4 5 6)                ⇒ (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                             ⇒ (5 6)
```

`if` *test consequent* [*alternate*] [syntax]

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

`set!` *variable expression* [syntax]

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

`cond` *clause clause* . . . [library syntax]

Bigloo considers `else` as a keyword. It thus ignores clauses following an `else`-clause.

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

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

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

`case` *key clause clause* . . . [library syntax]

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

`and` *test* . . . [library syntax]

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

`and-let*` *test* . . . [bigloo syntax]

```
(and-let* ((x 1) (y 2)) (cons x y))   ⇒ (1 . 2)
(and-let* ((x 1) (z #f)) x)           ⇒ #f

(and-let* ((my-list (compute-list)) ((not (null? my-list))))
          (do-something my-list))

(define (look-up key alist)
  (and-let* ((x (assq key alist))) (cdr x)))
```

```
(or (and-let* ((c (read-char))
               ((not (eof-object? c))))
              (string-set! some-str i c)
              (set! i (+ 1 i)))
```

`or` *test ...* [library syntax]

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

`let` [*name*] (*binding ...*) *body* [library syntax]

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

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

(let loop ((l '(1 2 3)))
   (if (null? l)
       '()
       (cons (+ 1 (car l))
             (loop (cdr l)))))      ⇒ (2 3 4)
```

If a *binding* is a symbol, then, it introduces a variable bound to the `#unspecified` value.

```
(let (x)
   x)                               ⇒ #unspecified
```

Bigloo's named let differs from R5Rs named let because *name* is bound in *binding*. That is,

```
(let ((l 'a-symbol))
  (let l ((x l))
     x))                            ⇒ #<procedure>
```

while R5Rs states that,

```
(let ((l 'a-symbol))
  (let l ((x l))
     x))                            ⇒ a-symbol
```

`let*` (*binding ...*) *body* [library syntax]

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

`letrec` (*binding ...*) *body* [library syntax]

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

```
                #f
                (even? (- n 1))))))
  (even? 88))
                                          ⇒ #t
```

`letrec*` (*binding ...*) *body* [bigloo syntax]

Each binding has the form

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

Each `<init>` is an expression.Any variable must not appear more than once in the `<variable>`s.

The `<variable>`s are bound to fresh locations, each <variable> is assigned in left-to-right order to the result of evaluating the corresponding `<init>`, the `<body>` is evaluated in the resulting environment, and the values of the last expression in <body> are returned. Despite the left-to-right evaluation and assignment order, each binding of a <variable> has the entire letrec* expression as its region, making it possible to define mutually recursive procedures.

Examples:

```
(letrec* ((x 1)
          (f (lambda (y) (+ x y))))
   (f 3))
                                          ⇒ 4
(letrec* ((p (lambda (x)
               (+ 1 (q (- x 1)))))
          (q (lambda (y)
               (if (zero? y)
                   0
                   (+ 1 (p (- y 1))))))
          (x (p 5))
          (y x))
  y)
                                          ⇒ 5
```

It must be possible to evaluate each `<init>` without assigning or referring to the value of the corresponding <variable> or the `<variable>` of any of the bindings that follow it in <bindings>. Another restriction is that the continuation of each `<init>` should not be invoked more than once.

`labels` ((*name* (*arg ...*) *body*) *...*) *body* [bigloo syntax]

The syntax is similar to the Common Lisp one [Steele90], where created bindings are immutable.

```
(labels ((loop (f l acc)
               (if (null? l)
                   (reverse! acc)
                   (loop f (cdr l) (cons (f (car l)) acc)))))
  (loop (lambda (x) (+ 1 x)) (list 1 2 3) '()))
  ⇒ (2 3 4)
```

`begin` *expression expression ...* [library syntax]

```
(define x 0)

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

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

`do` ((*variable init step*) ...) (*test expression ...*) *body* [library syntax]

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

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

`delay` *expression* [library syntax]

`quasiquote` *template* [syntax]

‘ *template* [syntax]

```
‘(list ,(+ 1 2) 4)                    ⇒ (list 3 4)
(let ((name 'a)) ‘(list ,name ',name))
          ⇒ (list a (quote a))
‘(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
          ⇒ (a 3 4 5 6 b)
‘((‘foo' ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
          ⇒ ((foo 7) . cons)
‘#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
          ⇒ #(10 5 2 4 3 8)
‘(a ‘(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ⇒ (a ‘(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  ‘(a ‘(b ,,name1 ,',name2 d) e))
          ⇒ (a ‘(b ,x ,'y d) e)
(quasiquote (list (unquote (+ 1 2)) 4))
          ⇒ (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
          ⇒ ‘(list ,(+ 1 2) 4)
    i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

3.1.3 Definitions

Global bindings are introduced by the `define` form:

`define` *variable expression* [syntax]

`define` (*variable arg ...*) *body* [syntax]

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                              ⇒ 6
(define first car)
(first '(1 2))                        ⇒ 1
```

See Section 3.1.3 [r5rs.info], page 22, for more details. The Bigloo module language (See Section 2.2 [Module Declaration], page 7) enables *exports* and *imports* of global definitions.

4 DSSSL support

Bigloo supports extensions for the Dsssl expression language [Dsssl96]:

- Keywords. Bigloo supports full Dsssl keywords.
- Named constants. Bigloo implements the three Dsssl named constants: `#!optional`, `#!rest` and `#!key`.
- Dsssl formal argument lists.

In addition, Bigloo extends DSSSL formal parameters. It supports `#!rest` argument following `!key` arguments. In that case, the `#!rest` formal parameter is bound to the list of non-keyword values.

4.1 DSSSL formal argument lists

Dsssl formal argument lists are defined by the following grammar:

```
<formal-argument-list> ↦ <required-formal-argument>*
  [(#!optional <optional-formal-argument>*)]
  [(#!rest <rest-formal-argument>)]
  [(#!key <key-formal-argument>*) (#!rest <rest-formal-argument>?)]
<required-formal-argument> ↦ <ieee-ident>
<optional-formal-argument> ↦ <ieee-ident>
     | (<ieee-ident> <initializer>)
<rest-formal-argument> ↦ <ieee-ident>
<key-formal-argument> ↦ <ieee-ident>
     | (<ieee-ident> <initializer>)
<initializer> ↦ <expr>
```

When a procedure is applied to a list of actual arguments, the formal and actual arguments are processed from left to right as follows:

- *Variables* in *required-formal-arguments* are bound to successive actual arguments starting with the first actual argument. It shall be an error if there are fewer actual arguments than *required-formal-arguments*.
- Next, *variables* in *optional-formal-arguments* are bound to any remaining actual arguments. If there are fewer remaining actual arguments than *optional-formal-arguments*, then variables are bound to the result of the evaluation of *initializer*, if one was specified, and otherwise to `#f`. The *initializer* is evaluated in an environment in which all previous optional formal arguments have been bound.
- If there is a *rest-formal-argument*, then it is bound to a list of all remaining actual arguments. The remaining actual arguments are also eligible to be bound to *keyword-formal-arguments*. If there is no *rest-formal-argument* and there are no *keyword-formal-arguments*, the it shall be an error if there are any remaining actual arguments.
- If `#!key` was specified in the *formal-argument-list*, there shall be an even number of remaining actual arguments. These are interpreted as a series of pairs, where the first member of each pair is a keyword specifying the argument name, and the second is the corresponding value. It shall be an error if the first member of a pair is not a keyword. It shall be an error if the argument name is not the same as a variable in a *keyword-formal-argument*, unless there is a *rest-formal-argument*. If the same

argument name occurs more than once in the list of actual arguments, then the first value is used. If there is no actual argument for a particular *keyword-formal-argument*, then the variable is bound to the result of evaluating *initializer* if one was specified, and otherwise `#f`. The *initializer* is evaluated in an environment in which all previous formal key arguments have been bound.

- If `#!rest` was specified in the *formal-argument-list after* a `#!key` formal parameter, it is bound to the list of optional *non-keyword* arguments.

It shall be an error for an <ieee-ident> to appear more than once in a *formal-argument-list*.

Example:

```
((lambda (x y #!rest z) z)
 3 4 5 6)                    ⇒ (5 6)
((lambda (x y #!optional z #!rest r #!key i (j 1))
    (list x y z i: i j: j))
 3 4 5 i: 6 i: 7)            ⇒ (3 4 5 i: 6 j: 1)
((lambda (x y #!optional z #!key i (j 1) #!rest r)
    (list x y z i: i j: j r))
 3 4 5 i: 6 i: 7 8 9)        ⇒ (3 4 5 i: 6 j: 1 (8 9))
```

4.2 Modules and DSSSL formal argument lists

Functions using Dsssl formal argument lists can be exported or imported in the same way as all regular Bigloo functions. When exporting such a Dsssl function the exact prototype of the function must be duplicated in the export clause. That is, for instance, the exportation prototype for the function:

```
(define (foo x y #!optional z #!key i (j 1)) ...)
```

is:

```
(export (foo x y #!optional z #!key i (j 1)))
```

5 Standard Library

This chapter presents the Bigloo standard library. Bigloo is mostly R5RS compliant but it proposes many extensions to this standard. In a first section (Section 5.1 [Scheme Library], page 25) the Bigloo R5RS support is presented. This section also contains various function that are not standard (for instance, various functions used to manage a file system). Then, in the following sections (Section 5.3 [Serialization], page 70, Section 5.4 [Bit Manipulation], page 72, and Section 5.7 [System Programming], page 76, Bigloo specific extensions are presented. Bigloo input and output facilities constitute a large superset of the standard Scheme definition. For this reason they are presented in a separate section (Section 5.2 [Input and Output], page 54).

5.1 Scheme Library

When the definition of a procedure or a special form is the same in Bigloo and Scheme, we just mention its name; otherwise, we explain it and qualify it as a "bigloo procedure".

5.1.1 Booleans

The standard boolean objects are `#t` and `#f`. Note: the empty list is true.

`not` *obj* [library procedure]

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

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

`boolean?` *obj* [library procedure]

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

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

5.1.2 Equivalence predicates

`eqv?` *obj1 obj2* [procedure]
`eq?` *obj1 obj2* [procedure]

`eqv?` and `eq?` are equivalent in Bigloo.

```
(eq? 'a 'a)                 ⇒  #t
(eq? '(a) '(a))             ⇒  unspecified
(eq? (list 'a) (list 'a))   ⇒  #f
(eq? "a" "a")               ⇒  unspecified
(eq? "" "")                 ⇒  unspecified
(eq? '() '())               ⇒  #t
(eq? 2 2)                   ⇒  unspecified
(eq? #\A #\A)               ⇒  unspecified
(eq? car car)               ⇒  #t
(let ((n (+ 2 3)))
  (eq? n n))                ⇒  unspecified
```

```
(let ((x '(a)))
  (eq? x x))                            ⇒  #t
(let ((x '#()))
  (eq? x x))                            ⇒  #t
(let ((p (lambda (x) x)))
  (eq? p p))                            ⇒  #t
```

Since Bigloo implements `eqv?` as `eq?`, the behavior is not always conforming to R5RS.

```
(eqv? 'a 'a)                            ⇒  #t
(eqv? 'a 'b)                            ⇒  #f
(eqv? 2 2)                              ⇒  #t
(eqv? '() '())                          ⇒  #t
(eqv? 100000000 100000000)              ⇒  #t
(eqv? (cons 1 2) (cons 1 2))            ⇒  #f
(eqv? (lambda () 1)
      (lambda () 2))                    ⇒  #f
(eqv? #f 'nil)                          ⇒  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                           ⇒  unspecified
```

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

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

(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                           ⇒  #t
(eqv? (gen-counter) (gen-counter))
                                        ⇒  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                           ⇒  #t
(eqv? (gen-loser) (gen-loser))
                                        ⇒  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                                        ⇒  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
                                        ⇒  #f
(eqv? '(a) '(a))                        ⇒  unspecified
(eqv? "a" "a")                          ⇒  unspecified
```

```
(eqv? '(b) (cdr '(a b)))              ⇒  unspecified
(let ((x '(a)))
  (eqv? x x))                         ⇒  #t
```

`equal?` *obj1 obj2* [library procedure]

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

See Section "r5rs.info" in `R5RS`, for more details.

5.1.3 Pairs and lists

The form `()` is *illegal*.

`pair?` *obj* [procedure]

`cons` *a d* [procedure]

`pair-or-null?` *obj* [bigloo procedure]

Returns `#t` if *obj* is either a pair or the empty list. Otherwise it returns `#f`.

`car` *pair* [procedure]
`cdr` *pair* [procedure]
`set-car!` *pair obj* [procedure]
`set-cdr!` *pair obj* [procedure]

`caar` *pair* [library procedure]
`cadr` *pair* [library procedure]
`cadar` *pair* [library procedure]
`caadr` *pair* [library procedure]
`caaar` *pair* [library procedure]
`caddr` *pair* [library procedure]
`cadar` *pair* [library procedure]
`cdddar` *pair* [library procedure]
`cddddr` *pair* [library procedure]

`null?` *obj* [library procedure]
`list?` *obj* [library procedure]
`list` *obj ...* [library procedure]
`length` *list* [library procedure]
`append` *list ...* [library procedure]
`append!` *list ...* [bigloo procedure]

A destructive append.

`reverse` *list* [library procedure]
`reverse!` *list* [bigloo procedure]

A destructive reverse.

`list-ref` *list k* [library procedure]
`take` *list k* [library procedure]
`drop` *list k* [library procedure]
`list-tail` *list k* [library procedure]

`list-ref` returns the *k* element of the list.

`take` returns a new list made of the first *k* element of the list.

`Drop` and `list-tail` returns the sublist of *list* obtained by omitting the first *k* elements.

`last-pair` *list* [bigloo procedure]

Returns the last pair in the nonempty, possibly improper, *list*.

`memq` *obj list* [library procedure]
`memv` *obj list* [library procedure]
`member` *obj list* [library procedure]
`assq` *obj alist* [library procedure]
`assv` *obj alist* [library procedure]
`assoc` *obj alist* [library procedure]
`remq` *obj list* [bigloo procedure]

Returns a new list which is a copy of *list* with all items `eq?` to *obj* removed from it.

`remq!` *obj list* [bigloo procedure]

Same as `remq` but in a destructive way.

`delete` *obj list* [*eq equal?*] [bigloo procedure]

Returns a new list which is a copy of *list* with all items `equal?` to *obj* deleted from it.

`delete!` *obj list* [*eq equal?*] [bigloo procedure]

Same as `delete` but in a destructive way.

`cons*` *obj . . .* [bigloo procedure]

Returns an object formed by consing all arguments together from right to left. If only one *obj* is supplied, that *obj* is returned.

`every` *fun clist1 clist2 ...* [bigloo procedure]

Applies the function *fun* across the lists, returning the last non-false if the function returns non-false on every application. If non-false, the result of `every` is the last value returned by the last application of *fun*.

```
(every < '(1 2 3) '(2 3 4))          ⇒ #t
(every < '(1 2 3) '(2 3 0))          ⇒ #f
```

`any` *fun clist1 clist2 ...* [bigloo procedure]

Applies the function *fun* across the lists, returning non-false if the function returns non-false for at least one application. If non-false, the result of `any` is the first non-false value returned by *fun*.

```
(any < '(1 2 3) '(2 3 4))          ⇒ #t
(any < '(1 2 3) '(2 3 0))          ⇒ #t
```

`find` *pred clist* [bigloo procedure]

Return the first element of *clist* that satisfies predicate *pred*; false if no element does.

```
(find even? '(3 1 4 1 5 9))          ⇒ 4
```

Note that `find` has an ambiguity in its lookup semantics – if find returns `#f`, you cannot tell (in general) if it found a `#f` element that satisfied *pred*, or if it did not find any element at all. In many situations, this ambiguity cannot arise – either the list being searched is known not to contain any `#f` elements, or the list is guaranteed to have an element satisfying pred. However, in cases where this ambiguity can arise, you should use find-tail instead of find – find-tail has no such ambiguity:

```
(cond ((find-tail pred lis) => (lambda (pair) ...)) ; Handle (CAR PAIR)
      (else ...)) ; Search failed.
```

`find-tail` *pred clist* [bigloo procedure]

Return the first pair of *clist* whose car satisfies *pred*. If no pair does, return false.

`find-tail` can be viewed as a general-predicate variant of the member function.

Examples:

```
(find-tail even? '(3 1 37 -8 -5 0 0)) ⇒ (-8 -5 0 0)
(find-tail even? '(3 1 37 -5)) ⇒ #f

;; MEMBER X LIS:
(find-tail (lambda (elt) (equal? x elt)) lis)

In the circular-list case, this procedure "rotates" the list.
```

`reduce` *f ridentity list* [bigloo procedure]

If *list* if null returns *ridentity*, if *list* has one element, returns that element. Otherwise, returns *f* applied to the first element of the *list* and to `reduce` of the rest of the list.

Examples:

```
(reduce max 0 l) ≡ (apply max l)
```

`make-list` *n* [*fill*] [bigloo procedure]

Returns an *n*-element list, whose elements are all the value *fill*. If the *fill* argument is not given, the elements of the list may be arbitrary values.

```
(make-list 4 'c)                     ⇒ (c c c c)
```

`list-tabulate` *n init-proc* [bigloo procedure]

Returns an *n*-element list. Element i of the list, where 0 <= i < *n*, is produced by `(init-proc i)`. No guarantee is made about the dynamic order in which *init-proc* is applied to these indices.

```
(list-tabulate 4 values)             ⇒ (0 1 2 3)
```

`list-split` *list n* [*filler*] [bigloo procedure]
`list-split` *list n* [*filler*] [bigloo procedure]

Split a *list* into a list of lists of length *n*. Last smaller list is filled with *filler*.

```
(list-split '(1 2 3 4 5 6 7 8) 3 0) ⇒ ((1 2 3) (4 5 6) (7 8 0))
(list-split (iota 10) 3)            ⇒ ((0 1 2) (3 4 5) (6 7 8) (9))
(list-split (iota 10 3) '-1)      ⇒ ((0 1 2) (3 4 5) (6 7 8) (9 -1 -1))
```

`iota` *count* [*start step*] [bigloo procedure]

Returns a list containing the elements

```
(start start+step ... start+(count-1)*step)
```

The *start* and *step* parameters default to `0` and `1`, respectively. This procedure takes its name from the APL primitive.

```
(iota 5) ⇒ (0 1 2 3 4)
(iota 5 0 -0.1) ⇒ (0 -0.1 -0.2 -0.3 -0.4)
```

`list-copy` *l* [bigloo procedure]
`tree-copy` *l* [bigloo procedure]

The function `list-copy` copies the spine of the of the list. The function `tree-copy` recursively copies its arguments, descending only into the list cells.

`delete-duplicates` *list* [*eq equal?*] [bigloo procedure]
`delete-duplicates!` *list* [*eq equal?*] [bigloo procedure]

`delete-duplicates` removes duplicate elements from the *list* argument. If there are multiple equal elements in the argument list, the result list only contains the first or leftmost of these elements in the result. The order of these surviving elements is the same as in the original list – `delete-duplicates` does not disorder the list (hence it is useful for "cleaning up" association lists).

The *equal* parameter is used to compare the elements of the list; it defaults to `equal?`. If x comes before y in list, then the comparison is performed (= x y). The comparison procedure will be used to compare each pair of elements in list no more than once; the order in which it is applied to the various pairs is not specified.

`delete-duplicates` is allowed to share common tails between argument and result lists – for example, if the list argument contains only unique elements, it may simply return exactly this list.

See Section "r5rs.info" in `R5RS`, for more details.

5.1.4 Symbols

Symbols are case sensitive and the reader is case sensitive too. So:

```
(eq? 'foo 'FOO) ⇒ #f
(eq? (string->symbol "foo") (string->symbol "FOO")) ⇒ #f
```

Symbols may contain special characters (such as #\Newline or #\Space). Such symbols that have to be read must be written: `|[^]+|`. The function `write` uses that notation when it encounters symbols containing special characters.

```
(write 'foo) ⇒ foo
(write 'Foo) ⇒Foo
(write '|foo bar|) ⇒ |foo bar|
```

`symbol?` *obj* [procedure]
`symbol->string` *symbol* [procedure]

Returns the name of the symbol as a string. Modifying the string result of `symbol->string` could yield incoherent programs. It is better to copy the string before any physical update. For instance, don't write:

```
(string-downcase! (symbol->string 'foo))
```

See Section "r5rs.info" in `R5RS`, for more details.

but prefer:

```
(string-downcase (symbol->string 'foo))
```

`string->symbol` *string* [procedure]
`string->symbol-ci` *string* [bigloo procedure]
`symbol-append` *symbol . . .* [bigloo procedure]

`String->symbol` returns a symbol whose name is *string*. `String->symbol` respects the case of *string*. `String->symbol-ci` returns a symbol whose name is `(string-upcase` *`string`*`)`. `Symbol-append` returns a symbol whose name is the concatenation of all the *symbol*'s names.

`gensym` [*obj*] [bigloo procedure]

Returns a new fresh symbol. If *obj* is provided and is a string or a symbol, it is used as prefix for the new symbol.

`genuuid` [bigloo procedure]

Returns a string containing a new fresh *Universal Unique Identifier* (see `http://fr.wikipedia.org/wiki/Universal_Unique_Identifier`).

`symbol-plist` *symbol-or-keyword* [bigloo procedure]

Returns the property-list associated with *symbol-or-keyword*.

`getprop` *symbol-or-keyword key* [bigloo procedure]

Returns the value that has the key `eq?` to *key* from the *symbol-or-keyword*'s property list. If there is no value associated with *key* then `#f` is returned.

`putprop!` *symbol-or-keyword key val* [bigloo procedure]

Stores *val* using *key* on *symbol-or-keyword*'s property list.

`remprop!` *symbol-or-keyword key* [bigloo procedure]

Removes the value associated with *key* in the *symbol-or-keyword*'s property list. The result is unspecified.

Here is an example of properties handling:

```
(getprop 'a-sym 'a-key)       ⇒ #f
(putprop! 'a-sym 'a-key 24)
(getprop 'a-sym 'a-key)       ⇒ 24
(putprop! 'a-sym 'a-key2 25)
(getprop 'a-sym 'a-key)       ⇒ 24
(getprop 'a-sym 'a-key2)      ⇒ 25
(symbol-plist 'a-sym)         ⇒ (a-key2 25 a-key 24)
(remprop! 'a-sym 'a-key)
(symbol-plist 'a-sym)         ⇒ (a-key2 25)
(putprop! 'a-sym 'a-key2 16)
(symbol-plist 'a-sym)         ⇒ (a-key2 16)
```

5.1.5 Keywords

Keywords constitute an extension to Scheme required by Dsssl [Dsssl96]. Keywords syntax is either <ident>: or :<ident>.

Keywords are autoquote and case sensitive. So

```
(eq? toto: TOTO:) ⇒ #f
```

The colon character (`:`) does not belong to they keyword. Hence

```
(eq? toto: :toto) ⇒ #t
```

`keyword?` *obj* [bigloo procedure]
`keyword->string` *keyword* [bigloo procedure]
`string->keyword` *string* [bigloo procedure]
`keyword->symbol` *keyword* [bigloo procedure]
`symbol->keyword` *symbol* [bigloo procedure]

5.1.6 Numbers

Bigloo has only three kinds of numbers: fixnum, long fixnum and flonum. Operations on complexes and rationals are not implemented but for compatibility purposes, the functions `complex?` and `rational?` exist. (In fact, `complex?` is the same as `number?` and `rational?` is the same as `real?` in Bigloo.) The accepted prefixes are `#b`, `#o`, `#d`, `#x`, `#e`, `#ex`, `#l`, `#lx`, `#z`, and `#zx`. For each generic arithmetic procedure, Bigloo provides two specialized procedures, one for fixnums and one for flonums. The names of these two specialized procedures is the name of the original one suffixed by `fx` (fixnum), `fl` (flonum), `elong` (exact C long), `llong` (exact C long long), and `bx` (big integer). A fixnum has the size of a C `long` minus 2 bits. A flonum has the size of a C `double`. An elong has the size of a C long. An llong has the size of a C long long. A big integer has an unbounded size.

`number?` *obj* [procedure]
`real?` *obj* [procedure]
`integer?` *obj* [procedure]
`complex?` *x* [bigloo procedure]
`rational?` *x* [bigloo procedure]

`fixnum?` *obj* [bigloo procedure]
`flonum?` *obj* [bigloo procedure]
These two procedures are type checkers on types `integer` and `real`.

`elong?` *obj* [bigloo procedure]
`llong?` *obj* [bigloo procedure]
The `elong?` procedures is a type checker for "hardware" integers, that is integers that have the very same size has the host platform permits (e.g., 32 bits or 64 bits integers). The `llong?` procedure is a type checker for "hardware" long long integers. Exact integers literal are introduced with the special `#e` and `#ex` prefixes. Exact long integers literal are introduced with the special `#l` and `#lx` prefixes.

`bignum?` *obj* [bigloo procedure]
This type checker tests if its argument is a big integer.

`make-elong` *int* [bigloo procedure]
`make-llong` *int* [bigloo procedure]
Create an exact fixnum integer from the fixnum value *int*.

`minvalfx` [bigloo procedure]
`maxvalfx` [bigloo procedure]
`minvalelong` [bigloo procedure]
`maxvalelong` [bigloo procedure]
`minvalllong` [bigloo procedure]
`maxvalllong` [bigloo procedure]
Returns the minimal value (respectively the maximal value) for fix integers.

`exact?` *z* [procedure]
`inexact?` *z* [procedure]

`zero?` *z* [library procedure]
`positive?` *z* [library procedure]
`negative?` *z* [library procedure]
`odd?` *n* [library procedure]
`even?` *n* [library procedure]
`zerofx?` *z* [library procedure]
`positivefx?` *z* [library procedure]
`negativefx?` *z* [library procedure]
`oddfx?` *n* [library procedure]
`evenfx?` *n* [library procedure]
`zerofl?` *z* [library procedure]
`positivefl?` *z* [library procedure]
`negativefl?` *z* [library procedure]
`oddfl?` *n* [library procedure]
`evenfl?` *n* [library procedure]
`zeroelong?` *z* [library procedure]
`positiveelong?` *z* [library procedure]
`negativeelong?` *z* [library procedure]
`oddelong?` *n* [library procedure]
`evenelong?` *n* [library procedure]
`zerollong?` *z* [library procedure]
`positivellong?` *z* [library procedure]
`negativellong?` *z* [library procedure]
`oddllong?` *n* [library procedure]
`evenllong?` *n* [library procedure]
`zerobx?` *z* [library procedure]
`positivebx?` *z* [library procedure]
`negativebx?` *z* [library procedure]
`oddbx?` *n* [library procedure]
`evenbx?` *n* [library procedure]

`min` *x1 x2 . . .* [library procedure]
`max` *x1 x2 . . .* [library procedure]
`minfx` *x1 x2 . . .* [bigloo procedure]
`maxfx` *x1 x2 . . .* [bigloo procedure]
`minfl` *x1 x2 . . .* [bigloo procedure]
`maxfl` *x1 x2 . . .* [bigloo procedure]
`minbx` *x1 x2 . . .* [bigloo procedure]
`maxbx` *x1 x2 . . .* [bigloo procedure]

`=` *z1 z2 z3 . . .* [procedure]
`=fx` *i1 i2* [bigloo procedure]
`=fl` *r1 r2* [bigloo procedure]
`=elong` *r1 r2* [bigloo procedure]
`=llong` *r1 r2* [bigloo procedure]
`=bx` *r1 r2* [bigloo procedure]

`<` *z1 z2 z3 . . .* [procedure]
`<fx` *i1 i2* [bigloo procedure]
`<fl` *r1 r2* [bigloo procedure]
`<elong` *r1 r2* [bigloo procedure]
`<lllong` *r1 r2* [bigloo procedure]
`<bx` *r1 r2* [bigloo procedure]
`>` *z1 z2 z3 . . .* [procedure]
`>fx` *i1 i2* [bigloo procedure]
`>fl` *r1 r2* [bigloo procedure]
`>elong` *r1 r2* [bigloo procedure]
`>lllong` *r1 r2* [bigloo procedure]
`>bx` *r1 r2* [bigloo procedure]
`<=` *z1 z2 z3 . . .* [procedure]
`<=fx` *i1 i2* [bigloo procedure]
`<=fl` *r1 r2* [bigloo procedure]
`<=elong` *r1 r2* [bigloo procedure]
`<=llong` *r1 r2* [bigloo procedure]
`<=bx` *r1 r2* [bigloo procedure]
`>=` *z1 z2 z3 . . .* [procedure]
`>=fx` *i1 i2* [bigloo procedure]
`>=fl` *r1 r2* [bigloo procedure]
`>=elong` *r1 r2* [bigloo procedure]
`>=llong` *r1 r2* [bigloo procedure]
`>=bx` *r1 r2* [bigloo procedure]

`+` *z . . .* [procedure]
`+fx` *i1 i2* [bigloo procedure]
`+fl` *r1 r2* [bigloo procedure]
`+elong` *r1 r2* [bigloo procedure]
`+llong` *r1 r2* [bigloo procedure]
`+bx` *r1 r2* [bigloo procedure]
`*` *z . . .* [procedure]
`*fx` *i1 i2* [bigloo procedure]
`*fl` *r1 r2* [bigloo procedure]
`*elong` *r1 r2* [bigloo procedure]
`*llong` *r1 r2* [bigloo procedure]
`*bx` *r1 r2* [bigloo procedure]
`-` *z* [procedure]
`-` *z1 z2 . . .* [procedure]
`-fx` *i1 i2* [bigloo procedure]
`-fl` *r1 r2* [bigloo procedure]
`-elong` *r1 r2* [bigloo procedure]
`-llong` *r1 r2* [bigloo procedure]
`-bx` *r1 r2* [bigloo procedure]
`negfx` *i* [bigloo procedure]
`negfl` *r* [bigloo procedure]
`negelong` *r* [bigloo procedure]

`negllong` *r* [bigloo procedure]
`negbx` *r* [bigloo procedure]

These two functions implement the unary function `-`.

`/` *z1 z2* [procedure]
`/` *z1 z2 . . .* [procedure]
`/fx` *i1 i2* [bigloo procedure]
`/fl` *r1 r2* [bigloo procedure]
`/elong` *r1 r2* [bigloo procedure]
`/llong` *r1 r2* [bigloo procedure]
`/bx` *r1 r2* [bigloo procedure]

`abs` *z* [library procedure]
`absfl` *z* [bigloo procedure]
`quotient` *z1 z2* [procedure]
`quotientelong` *z1 z2* [procedure]
`quotientllong` *z1 z2* [procedure]
`remainder` *z1 z2* [procedure]
`remainderelong` *z1 z2* [procedure]
`remainderllong` *z1 z2* [procedure]
`remainderfl` *z1 z2* [procedure]
`modulo` *z1 z2* [procedure]
`gcd` *z . . .* [procedure]
`lcm` *z . . .* [procedure]
`floor` *z* [procedure]
`floorfl` *z* [procedure]
`ceiling` *z* [procedure]
`ceilingfl` *z* [procedure]
`truncate` *z* [procedure]
`truncatefl` *z* [procedure]
`round` *z* [procedure]
`roundfl` *z* [procedure]

`random` *z* [bigloo procedure]
`randomfl` [bigloo procedure]
`randombx` *z* [bigloo procedure]
`seed-random!` *z* [bigloo procedure]

the `random` function returns a pseudo-random integer between 0 and *z*.

If no seed value is provided, the `random` function is automatically seeded with a value of 1.

The function `randomfl` returns a double in the range [0..1].

`exp` *z* [procedure]
`expfl` *z* [procedure]
`log` *z* [procedure]
`logfl` *z* [procedure]
`sin` *z* [procedure]
`sinfl` *z* [procedure]

`cos` *z* [procedure]
`cosfl` *z* [procedure]
`tan` *z* [procedure]
`tanfl` *z* [procedure]
`asin` *z* [procedure]
`asinfl` *z* [procedure]
`acos` *z* [procedure]
`acosfl` *z* [procedure]
`atan` *z1 z2* [procedure]
`atanfl` *z1 z2* [procedure]
`sqrt` *z* [procedure]
`sqrtfl` *z* [procedure]
`expt` *z1 x2* [procedure]
`exptfl` *z1 x2* [procedure]

`exact->inexact` *z* [procedure]
`inexact->exact` *z* [procedure]
`number->string` *z* [procedure]
`integer->string` *i* [*radix 10*] [bigloo procedure]
`integer->string/padding` *i padding* [*radix 10*] [bigloo procedure]
`elong->string` *i* [*radix 10*] [bigloo procedure]
`llong->string` *i* [*radix 10*] [bigloo procedure]
`bignum->string` *i* [*radix 10*] [bigloo procedure]
`real->string` *z* [bigloo procedure]
`unsigned->string` *i* [*radix 16*] [bigloo procedure]

The function `integer->string/padding` converts its arguments into a string with a left padding filled of characters `0`.

```
(integer->string/padding 3 5)        ⇒ "00003"
```

The function `unsigned->string` only accepts the following radixes: `2`, `8`, and `16`. It converts its argument into an *unsigned* representation.

```
(unsigned->string 123 16)            ⇒ "7b"
(unsigned->string -123 16)           ⇒ "ffffff85"
```

`nanfl?` *z* [bigloo procedure]

Returns `#t` if the floating *z* is `not-a-number`. Returns `#f` otherwise.

`infinitefl?` *z* [bigloo procedure]
`finitefl?` *z* [bigloo procedure]

The predicate `infinitefl?` returns `#t` if the floating *z* is positive or negative infinite. Returns `#f` otherwise.

The predicate `finitefl?` is true if and only if `z` is finite.

`signbitfl` *z* [bigloo procedure]

Returns `0` is `z` is positive or null. Returns a positive integer otherwise.

`bignum->octet-string` *bignum* [bigloo procedure]

Returns a binary big-endian representation of the given bignum *bignum*.

```
(string-hex-extern (bignum->octet-string #zx1234567)) ⇒ "01234567"
```

`double->ieee-string` *z* [bigloo procedure]
`float->ieee-string` *z* [bigloo procedure]

Returns a big-endian representation of the given number.

`string->number` *string* [*radix 10*] [procedure]
`string->real` *string* [bigloo procedure]
`string->elong` *string radix* [bigloo procedure]
`string->llong` *string radix* [bigloo procedure]
`string->bignum` *string radix* [bigloo procedure]

Bigloo implements a restricted version of `string->number`. If *string* denotes a floating point number then, the only radix `10` may be send to `string->number`. That is:

```
(string->number "1243" 16)          ⇒ 4675
(string->number "1243.0" 16)        ⊣
# *** ERROR:bigloo:string->number
# Only radix '10' is legal for floating point number -- 16
(string->elong "234456353")         ⇒ #e234456353
```

In addition, `string->number` does not support radix encoded inside *string*. That is:

```
(string->number "#x1243")           ⇒ #f
```

`octet-string->bignum` *string* [bigloo procedure]

Counterpart to `bignum->octet-string`. Takes the bignum representation in big-endian format *string* and returns the corresponding bignum.

```
(octet-string->bignum (bignum->octet-string #z1234)) ⇒ #z1234
```

`ieee-string->double` *string* [bigloo procedure]
`ieee-string->float` *string* [bigloo procedure]

Convert the big-endian representations to their numeric values.

`fixnum->flonum` *i* [bigloo procedure]
`flonum->fixnum` *r* [bigloo procedure]
`elong->fixnum` *i* [bigloo procedure]
`fixnum->elong` *r* [bigloo procedure]
`llong->fixnum` *i* [bigloo procedure]
`fixnum->llong` *r* [bigloo procedure]
`elong->flonum` *i* [bigloo procedure]
`flonum->elong` *r* [bigloo procedure]
`llong->flonum` *i* [bigloo procedure]
`flonum->llong` *r* [bigloo procedure]

For efficiency, `string->real` and `string->integer` do not test whether the string can be read as a number. Therefore the result might be wrong if the string cannot be read as a number.

These last procedures implement the natural translation from and to fixnum, flonum, elong, and llong.

`double->llong-bits` *z* [bigloo procedure]
`float->int-bits` *z* [bigloo-procedure]

Returns the double-bits as a llong.

`llong-bits->double` *llong* [bigloo procedure]
`int-bits->float` *int* [bigloo procedure]

Converts the given llong bits to a double.

See Section "r5rs.info" in `R5RS`, for more details.

5.1.7 Characters

Bigloo knows named characters `#\alarm`, `#\backspace`, `#\delete`, `#\escape`, `#\tab`, `#\return`, and `#\null` in addition to the `#\space` and `#\newline` of R5RS.

A new alternate syntax exists for characters: `#a<ascii-code>` where `<ascii-code>` is the three digit decimal ASCII number of the character to be read. Thus, for instance, the character `#\space` can be written `#a032`. Bigloo also supports the R7Rs syntax `#\x<hex-code>`.

`char?` *obj* [procedure]

`char=?` *char1 char2* [procedure]
`char<?` *char1 char2* [procedure]
`char>?` *char1 char2* [procedure]
`char<=?` *char1 char2* [procedure]
`char>=?` *char1 char2* [procedure]
`char-ci=?` *char1 char2* [library procedure]
`char-ci<?` *char1 char2* [library procedure]
`char-ci>?` *char1 char2* [library procedure]
`char-ci<=?` *char1 char2* [library procedure]
`char-ci>=?` *char1 char2* [library procedure]

`char-alphabetic?` *char* [library procedure]
`char-numeric?` *char* [library procedure]
`char-whitespace?` *char* [library procedure]
`char-upper-case?` *char* [library procedure]
`char-lower-case?` *char* [library procedure]

`char->integer` *char* [procedure]
`integer->char` *i* [procedure]

`char-upcase` *char* [library procedure]
`char-downcase` *char* [library procedure]

5.1.8 UCS-2 Characters

UCS-2 Characters are two byte encoded characters. They can be read with the syntax: `#u<unicode>` where `<unicode>` is the four digit hexadecimal Unicode value of the character to be read. Thus, for instance, the character `#\space` can be written `#u0020`.

`ucs2?` *obj* [bigloo procedure]

`ucs2=?` *ucs2a ucs2b* [bigloo procedure]
`ucs2<?` *ucs2a ucs2b* [bigloo procedure]
`ucs2>?` *ucs2a ucs2b* [bigloo procedure]
`ucs2<=?` *ucs2a ucs2b* [bigloo procedure]

`ucs2>=?` *ucs2a ucs2b* [bigloo procedure]
`ucs2-ci=?` *ucs2a ucs2b* [bigloo procedure]
`ucs2-ci<?` *ucs2a ucs2b* [bigloo procedure]
`ucs2-ci>?` *ucs2a ucs2b* [bigloo procedure]
`ucs2-ci<=?` *ucs2a ucs2b* [bigloo procedure]
`ucs2-ci>=?` *ucs2a ucs2b* [bigloo procedure]

`ucs2-alphabetic?` *ucs2* [bigloo procedure]
`ucs2-numeric?` *ucs2* [bigloo procedure]
`ucs2-whitespace?` *ucs2* [bigloo procedure]
`ucs2-upper-case?` *ucs2* [bigloo procedure]
`ucs2-lower-case?` *ucs2* [bigloo procedure]

`ucs2->integer` *ucs2* [bigloo procedure]
`integer->ucs2` *i* [bigloo procedure]

`ucs2->char` *ucs2* [bigloo procedure]
`char->ucs2` *char* [bigloo procedure]

`ucs2-upcase` *ucs2* [bigloo procedure]
`ucs2-downcase` *ucs2* [bigloo procedure]

5.1.9 Strings

There are three different syntaxes for strings in Bigloo: traditional, foreign or Unicode. The traditional syntax for strings may conform to the Revised Report, see Section "r5rs.info" in `R5RS`. With the foreign syntax, C escape sequences are interpreted as specified by ISO-C. In addition, Bigloo's reader evaluate `\x??` sequence as an hexadecimal escape character. For Unicode syntax, see Section 5.1.10 [Unicode (UCS-2) Strings], page 44. Only the reader distinguishes between these three appearances of strings; i.e., there is only one type of string at evaluation-time. The regular expression describing the syntax for foreign string is: `#"([^"]|\")*"`. Escape characters are controlled by the parameter `bigloo-strict-r5rs-strings` (see Chapter 24 [Parameters], page 231).

The library functions for string processing are:

`string?` *obj* [procedure]

`string-null?` *s* [SRFI-13 procedure]
Is *s* an empty string?

`make-string` *k* [procedure]
`make-string` *k char* [procedure]
`string` *char ...* [library procedure]

`string-length` *string* [procedure]
`string-ref` *string k* [procedure]
`string-set!` *string k char* [procedure]

`string=?` *string1 string2* [library procedure]
This function returns `#t` if the *string1* and *string2* are made of the same characters. It returns `#f` otherwise.

`substring=?` *string1 string2 len* [bigloo procedure]

This function returns `#t` if *string1* and *string2* have a common prefix of size *len*.

```
(substring=? "abcdef" "ab9989898" 2)
   ⇒ #t
(substring=? "abcdef" "ab9989898" 3)
   ⇒ #f
```

`substring-at?` *string1 string2 offset* [*len*] [bigloo procedure]
`substring-ci-at?` *string1 string2 offset* [*len*] [bigloo procedure]

This function returns `#t` if *string2* is at position *offset* in the string `string1`. It returns `#f` otherwise.

```
(substring-at? "abcdefghij" "def" 3)
   ⇒ #t
(substring-at? "abcdefghij" "def" 2)
   ⇒ #f
(substring-at? "abcdefghij" "defz" 3)
   ⇒ #f
(substring-at? "abcdefghij" "defz" 3 3)
   ⇒ #t
```

`string-ci=?` *string1 string2* [library procedure]
`substring-ci=?` *string1 string2 len* [bigloo procedure]
`string<?` *string1 string2* [library procedure]
`string>?` *string1 string2* [library procedure]
`string<=?` *string1 string2* [library procedure]
`string>=?` *string1 string2* [library procedure]
`string-ci<?` *string1 string2* [library procedure]
`string-ci>?` *string1 string2* [library procedure]
`string-ci<=?` *string1 string2* [library procedure]
`string-ci>=?` *string1 string2* [library procedure]

`string-index` *string charset* [*start 0*] [*count -1*] [bigloo procedure]
`string-char-index` *string char* [*start 0*] [bigloo procedure]
`string-index-right` *string charset* [*start len-1*] [bigloo procedure]

Returns the first occurrence of a character of *char-or-set* in *string*. The argument *charset* is either a character or a string. If no character is found, `string-index` returns `#f` The argument *count*, if provided, is the number of characters to be scanned in the string.

`string-skip` *string charset* [*start 0*] [bigloo procedure]
`string-skip-right` *string charset* [*start len-1*] [bigloo procedure]

`string-skip` (resp. `string-skip-right`) searches through the *string* from the left (resp. right), returning the index of the first occurrence of a character which

- is not equal to c (if c is a character);
- is not in c (if c is a character set);
- does not satisfy the predicate c (if c is a procedure).

If no such index exists, the functions return false.

The start and end parameters specify the beginning and end indices of the search; the search includes the start index, but not the end index. Be careful of "fencepost" considerations: when searching right-to-left, the first index considered is end-1 whereas

when searching left-to-right, the first index considered is start. That is, the start/end indices describe a same half-open interval [start,end).

`string-contains` *string1 string2* [*start 0*] [bigloo procedure]
`string-contains-ci` *string1 string2* [*start 0*] [bigloo procedure]
Does string *string1* contain string *string2*?

Return the index in *string1* where *string2* occurs first as a substring, or false.

`string-contains-ci` is the case-insensitive variant. Case-insensitive comparison is done by case-folding characters with the operation:

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

`string-compare3` *string1 string2* [bigloo procedure]
`string-compare3-ci` *string1 string2* [bigloo procedure]
This function compares *string1* and *string2*. It returns a negative integer if *string1* < *string2*. It returns zero if the *string1* equal *string2*. It returns a positive integer if *string1* > *string2*.

`string-natural-compare3` *string1 string2* [*start1 0*] [*start2 0*] [bigloo procedure]
`string-natural-compare3-ci` *string1 string2* [*start1 0*] [*start2 0*] [bigloo procedure]
This function compares *string1* and *string2* according to a *natural string order*. It returns a negative integer if *string1* < *string2*. It returns zero if the *string1* equal *string2*. It returns a positive integer if *string1* > *string2*.

```
(string-natural-compare "foo" "foo")
   ⇒ 0
(string-natural-compare "foo0" "foo1")
   ⇒ -1
(string-natural-compare "foo1" "foo0")
   ⇒ 1
(string-natural-compare "rfc822.txt" "rfc1.txt")
   ⇒ -1
(string-natural-compare "rfc1.txt" "rfc2086.txt")
   ⇒ -1
(string-natural-compare "rfc2086.txt" "rfc1.txt")
   ⇒ 1
(string-natural-compare "rfc822.txt" "rfc2086.txt")
   ⇒ -1
(string-natural-compare "a0" "a1")
   ⇒ -1
(string-natural-compare "a1" "a1a")
   ⇒ -1
(string-natural-compare "a1a" "a1b")
   ⇒ -1
(string-natural-compare "a1b" "a2")
   ⇒ -1
(string-natural-compare "a2" "a10")
   ⇒ -1
(string-natural-compare "a10" "a20")
   ⇒ -1
(string-natural-compare "a2" "a20")
   ⇒ -1
(string-natural-compare "x2-g8" "x2-y7")
   ⇒ -1
```

```
(string-natural-compare "1.001" "1.002")
   ⇒ -1
(string-natural-compare "1.002" "1.010")
   ⇒ -1
(string-natural-compare "1.010"  "1.02")
   ⇒ 1
(string-natural-compare "1.02" "1.1")
   ⇒ -1
(string-natural-compare "1.1" "1.02")
   ⇒ 1
(string-natural-compare "1.02" "1.3")
   ⇒ -1
```

`substring` *string start* [*end*] [library procedure]

string must be a string, and *start* and *end* must be exact integers satisfying:

```
0 <= START <= END <= (string-length STRING)
```

The optional argument *end* defaults to `(string-length STRING)`.

`substring` returns a newly allocated string formed from the characters of *STRING* beginning with index *START* (inclusive) and ending with index *END* (exclusive).

```
(substring "abcdef" 0 5)
   ⇒ "abcde"
(substring "abcdef" 1 5)
   ⇒ "bcde"
```

`string-shrink!` *string end* [library procedure]

string must be a string, and *end* must be an exact integers satisfying:

```
0 <= END <= (string-length STRING)
```

`string-shrink!` returns a new string formed from the characters of *STRING* beginning with index 0 (inclusive) and ending with index *END* (exclusive). As much as possible `string-shrink!` changes the argument *string*. That is, as much as possible, and for the back-ends that enable it, *string-shrink!* operates a side effect on its argument.

```
(let ((s (string #\a #\b #\c #\d #\e)))
   (set! s (string-shrink! s 3))
   s)
   ⇒ "abc"
```

`string-append` *string ...* [library procedure]
`string->list` *string* [library procedure]
`list->string` *list* [library procedure]
`string-copy` *string* [library procedure]

`string-fill!` *string char* [bigloo procedure]

Stores *char* in every element of the given *string* and returns an unspecified value.

`string-downcase` *string* [bigloo procedure]

Returns a newly allocated version of string where each upper case letter is replaced by its lower case equivalent.

`string-upcase` *string* [bigloo procedure]

Returns a newly allocated version of string where each lower case letter is replaced by its upper case equivalent.

`string-capitalize` *string* [bigloo procedure]
Builds a newly allocated capitalized string.

`string-downcase!` *string* [bigloo procedure]
Physically downcases the *string* argument.

`string-upcase!` *string* [bigloo procedure]
Physically upcases the *string* argument.

`string-capitalize!` *string* [bigloo procedure]
Physically capitalized the *string* argument.

`string-for-read` *string* [bigloo procedure]
Returns a copy of *string* with each special character replaced by an escape sequence.

`blit-string!` *string1 o1 string2 o2 len* [bigloo procedure]
Fill string *string2* starting at position *o2* with *len* characters taken out of string *string1* from position *o1*.

```
(let ((s (make-string 20 #\-)))
(blit-string! "toto" 0 s 16 4)
s)
   ⇒ "----------------toto"
```

`string-replace` *string char1 char2* [bigloo procedure]
`string-replace!` *string char1 char2* [bigloo procedure]
Replace all the occurrence of *char1* by *char2* in *string*. The function `string-replace` returns a newly allocated string. The function `string-replace!` modifies its first argument.

`string-split` *string* [bigloo procedure]
`string-split` *string delimiters* [bigloo procedure]
Parses *string* and returns a list of tokens ended by a character of the *delimiters* string. If *delimiters* is omitted, it defaults to a string containing a space, a tabulation and a newline characters.

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

`string-cut` *string* [bigloo procedure]
`string-cut` *string delimiters* [bigloo procedure]
The function `string-cut` behaves as `string-split` but it introduces empty strings for consecutive occurrences of delimiters.

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

`string-delete` *string char/charset/pred s* [*start end*] [SRFI-13 procedure]
Filter the string *string*, retaining only those characters that are not equal to *char*, not present in *charset*, or not satisfying *pred*. This function returns a fresh string no larger than *end* - *start*.

`string-prefix-length` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]
`string-suffix-length` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]
`string-prefix-length-ci` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]
`string-suffix-length-ci` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]

Return the length of the longest common prefix/suffix of the two strings. For prefixes, this is equivalent to the "mismatch index" for the strings (modulo the starti index offsets).

The optional start/end indices restrict the comparison to the indicated substrings of s1 and s2.

`string-prefix?` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]
`string-suffix?` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]
`string-prefix-ci?` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]
`string-suffix-ci?` *s1 s2* [*start1 end1 start2 end2*] [SRFI-13 procedure]

Is s1 a prefix/suffix of s2?

The optional start/end indices restrict the comparison to the indicated substrings of s1 and s2.

`string-hex-intern` *string* [bigloo procedure]
`string-hex-intern!` *string* [bigloo procedure]

Converts an hexadecimal *string* of `n` characters into an actual string of `n/2` characters.

```
(string-hex-intern "4a4b4c") ⇒ "JKL"
```

`string-hex-extern` *string* [*start* [*end*]] [bigloo procedure]

Converts a *string* into a hexadecimal representation.

string must be a string, and *start* and *end* must be exact integers satisfying:

```
0 <= START <= END <= (string-length STRING)
```

The optional argument *start* default to 0. The optional argument *end* defaults to `(string-length STRING)`.

```
(string-hex-extern "JKL") ⇒ "4a4b4c"
```

5.1.10 Unicode (UCS-2) Strings

UCS-2 strings cannot be read by the standard reader but UTF-8 strings can. The special syntax for UTF-8 is described by the regular expression: `#u"([^]|\")*"`.

The library functions for Unicode string processing are:

`ucs2-string?` *obj* [bigloo procedure]

`make-ucs2-string` *k* [bigloo procedure]
`make-ucs2-string` *k char* [bigloo procedure]
`ucs2-string` *k ...* [bigloo procedure]

`ucs2-string-length` *s-ucs2* [bigloo procedure]
`ucs2-string-ref` *s-ucs2 k* [bigloo procedure]
`ucs2-string-set!` *s-ucs2 k char* [bigloo procedure]

`ucs2-string=?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string-ci=?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string<?` *s-ucs2a s-ucs2b* [bigloo procedure]

`ucs2-string>?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string<=?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string>=?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string-ci<?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string-ci>?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string-ci<=?` *s-ucs2a s-ucs2b* [bigloo procedure]
`ucs2-string-ci>=?` *s-ucs2a s-ucs2b* [bigloo procedure]

`ucs2-substring` *s-ucs2 start end* [bigloo procedure]
`ucs2-string-append` *s-ucs2 . . .* [bigloo procedure]
`ucs2-string->list` *s-ucs2* [bigloo procedure]
`list->ucs2-string` *chars* [bigloo procedure]
`ucs2-string-copy` *s-ucs2* [bigloo procedure]

`ucs2-string-fill!` *s-ucs2 char* [bigloo procedure]
Stores *char* in every element of the given s-ucs2 and returns an unspecified value.

`ucs2-string-downcase` *s-ucs2* [bigloo procedure]
Builds a newly allocated ucs2-string with lower case letters.

`ucs2-string-upcase` *s-ucs2* [bigloo procedure]
Builds a new allocated ucs2-string with upper case letters.

`ucs2-string-downcase!` *s-ucs2* [bigloo procedure]
Physically downcases the *s-ucs2* argument.

`ucs2-string-upcase!` *s-ucs2* [bigloo procedure]
Physically upcases the *s-ucs2* argument.

`ucs2-string->utf8-string` *s-ucs2* [bigloo procedure]
`utf8-string->ucs2-string` *string* [bigloo procedure]
Convert UCS-2 strings to (or from) UTF-8 encoded ascii strings.

`utf8-string?` *string* [*strict* `#f`] [bigloo procedure]
Returns `#t` if and only if the argument *string* is a well formed UTF-8 string. Otherwise returns `#f`.

If the optional argument *strict* is `#t`, half utf16-surrogates are rejected. The optional argument *strict* defaults to `#f`.

`ascii-string?` *string* [bigloo procedure]
Returns `#t` if and only if the argument *string* is only composed of ascii characters. Otherwise returns `#f`.

`utf8-string-encode` *string* [*strict* `#f`] [bigloo procedure]
Returns a copy of *string* where all the illegal UTF-8 prefix are replaced with the Unicode Replacement Character EF BF BD. The result is a well formed UTF-8 string.

`utf8-string-length` *string* [bigloo procedure]
Returns the number of characters of an UTF-8 string. It raises an error if the string is not a well formed UTF-8 string (i.e., it does satisfies the `utf8-string?` predicate.

`utf8-codepoint-length` *string* [bigloo procedure]
Returns the number of code points of an UTF-8 string. The code points length is the number of 16bits long values needed to encode the utf8 strings in utf16.

`utf8-string-ref` *string i* [bigloo procedure]
Returns the character (represented as an UTF-8 string) at the position *i* in *string*.

`utf8-substring` *string start* [*end*] [library procedure]
string must be a string, and *start* and *end* must be exact integers satisfying:

```
0 <= START <= END <= (string-length STRING)
```

The optional argument *end* defaults to `(utf8-string-length STRING)`.

`utf8-substring` returns a newly allocated string formed from the characters of *STRING* beginning with index *START* (inclusive) and ending with index *END* (exclusive).

If the argument *string* is not a well formed UTF-8 string an error is raised. Otherwise, the result is also a well formed UTF-8 string.

`iso-latin->utf8` *string* [bigloo procedure]
`iso-latin->utf8!` *string* [bigloo procedure]
`utf8->iso-latin` *string* [bigloo procedure]
`utf8->iso-latin!` *string* [bigloo procedure]
`utf8->iso-latin-15` *string* [bigloo procedure]
`utf8->iso-latin-15!` *string* [bigloo procedure]
Encode and decode iso-latin strings into utf8. The functions `iso-latin->utf8-string!`, `utf8->iso-latin!` and `utf8->iso-latin-15!` may return, as result, the string they receive as argument.

`cp1252->utf8` *string* [bigloo procedure]
`cp1252->utf8!` *string* [bigloo procedure]
`utf8->cp1252` *string* [bigloo procedure]
`utf8->cp1252!` *string* [bigloo procedure]
Encode and decode cp1252 strings into utf8. The functions `cp1252->utf8-string!` and `utf8->cp1252!` may return, as result, the string they receive as argument.

`8bits->utf8` *string table* [bigloo procedure]
`8bits->utf8!` *string table* [bigloo procedure]
`utf8->8bits` *string inv-table* [bigloo procedure]
`utf8->8bits!` *string inv-table* [bigloo procedure]
These are the general conversion routines used internally by `iso-latin->utf8` and `cp1252->utf8`. They convert any 8 bits *string* into its equivalent UTF-8 representation and vice versa.

The argument *table* should be either `#f`, which means that the basic (i.e., iso-latin-1) 8bits -> UTF-8 conversion is used, or it must be a vector of at least 127 entries containing strings of characters. This table contains the encodings for the 8 bits characters whose code range from 128 to 255.

The table is not required to be complete. That is, it is not required to give the whole character encoding set. Only the characters that need a non-iso-latin canonical representation must be given. For instance, the CP1252 table can be defined as:

```
(define cp1252
```

```
'#("\xe2\x82\xac" ;; 0x80
   ""             ;; 0x81
   "\xe2\x80\x9a" ;; 0x82
   "\xc6\x92"     ;; 0x83
   "\xe2\x80\x9e" ;; 0x84
   "\xe2\x80\xa6" ;; 0x85
   "\xe2\x80\xa0" ;; 0x86
   "\xe2\x80\xa1" ;; 0x87
   "\xcb\x86"     ;; 0x88
   "\xe2\x80\xb0" ;; 0x89
   "\xc5\xa0"     ;; 0x8a
   "\xe2\x80\xb9" ;; 0x8b
   "\xc5\x92"     ;; 0x8c
   ""             ;; 0x8d
   "\xc5\xbd"     ;; 0x8e
   ""             ;; 0x8f
   ""             ;; 0x90
   "\xe2\x80\x98" ;; 0x91
   "\xe2\x80\x99" ;; 0x92
   "\xe2\x80\x9c" ;; 0x93
   "\xe2\x80\x9d" ;; 0x94
   "\xe2\x80\xa2" ;; 0x95
   "\xe2\x80\x93" ;; 0x96
   "\xe2\x80\x94" ;; 0x97
   "\xcb\x9c"     ;; 0x98
   "\xe2\x84\xa2" ;; 0x99
   "\xc5\xa1"     ;; 0x9a
   "\xe2\x80\xba" ;; 0x9b
   "\xc5\x93"     ;; 0x9c
   ""             ;; 0x9d
   "\xc5\xbe"     ;; 0x9e
   "\xc5\xb8"))   ;; 0x9f
```

The argument *inv-table* is an inverse table that can be build from a table and using the function `inverse-utf8-table`.

`inverse-utf8-table` *vector* [procedure]
Inverse an UTF-8 table into an object suitable for `utf8->8bits` and `utf8->8bits!`.

5.1.11 Vectors

Vectors are not autoquoted objects.

`vector?` *obj* [procedure]

`make-vector` *k* [procedure]
`make-vector` *k obj* [procedure]
`vector` *obj ...* [library procedure]

`vector-length` *vector* [procedure]
`vector-ref` *vector k* [procedure]
`vector-set!` *vector k obj* [procedure]

`vector->list` *vector* [library procedure]
`list->vector` *list* [library procedure]

`vector-fill!` *vector obj* [*start* [*end*]] [library procedure]

Stores *obj* in every element of *vector*. For instance:

```
(let ((v (make-vector 5 #f)))
   (vector-fill! v #t)
   v)
```

`copy-vector` *vector len* [bigloo procedure]

Allocate a new vector of size *len* and fills it with the first *len* element of *vector*. The new length *len* may be bigger than the old vector length.

`vector-copy` *vector start end* [bigloo procedure]

vector must be a vector, and *start* and *end* must be exact integers satisfying:

```
0 <= START <= END <= (vector-length VECTOR)
```

`vector-copy` returns a newly allocated vector formed from the elements of *VECTOR* beginning with index *START* (inclusive) and ending with index *END* (exclusive).

```
(vector-copy '#(1 2 3 4) 0 4)
   ⇒ '#(1 2 3 4)
(vector-copy '#(1 2 3 4) 1 3)
   ⇒ '#(2 3)
```

`vector-copy!` *target tstart source* [*sstart* [*send*]] [bigloo procedure]

Copies a block of elements from *source* to *target*, both of which must be vectors, starting in target at *tstart* and starting in source at *sstart*, ending when *send* - *sstart* elements have been copied. It is an error for *target* to have a length less than `tstart + (send - sstart)`. *Sstart* defaults to `0` and *send* defaults to the length of *source*.

See Section 5.1.11 [r5rs.info], page 47, for more details.

`vector-append` *vector ...* [bigloo procedure]

Returns a newly allocated vector that contains all elements in order from the subsequent locations in vector ....

Examples:

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

`vector-for-each` *proc vector ...* [bigloo procedure]

Apply *proc* to all the elements of the *vectors*. The arity of *proc* must be the number of passed vectors. All vectors must have the same length. The procedure is applied from elements of index `0` to `(vector-length vector) - 1`.

`vector-map` *proc vector ...* [bigloo procedure]
`vector-map!` *proc vector ...* [bigloo procedure]

The function `vector-map` creates a new vector whose size the is the size of its argument *vector*. Each elements of the new vector is the result of apply *proc* to the corresponding elements of the initial vectors.

The function `vector-map!` modifies the elements of the argument *vector*.

`vector-shrink!` *vector end* [bigloo procedure]

Shrink a vector. The argument *vector* must be a vector, and *end* must be an exact integers satisfying:

```
0 <= END <= (vector-length STRING)
```

Shrink a vector. The resulting vector's len is the minimum value of `(vector-length vec)` and *nlen*.

`vector-shrink!` returns a new vector formed from the values of *VECTOR* beginning with index 0 (inclusive) and ending with index *END* (exclusive). As much as possible `vector-shrink!` changes the argument *vector*. That is, as much as possible, and for the back-ends that enable it, *vector-shrink!* operates a side effect on its argument.

5.1.12 Homogeneous Vectors (SRFI-4)

Bigloo fully supports SRFI-4 specification of homogeneous vectors (see `http://srfi.schemers.org/srfi-4/srfi-4.html`).

Each homogeneous vector is represented by a Bigloo type. That is:

- `::s8vector` signed exact integer in the range -(2^7) to (2^7)-1
- `::u8vector` unsigned exact integer in the range 0 to (2^8)-1
- `::s16vector` signed exact integer in the range -(2^15) to (2^15)-1
- `::u16vector` unsigned exact integer in the range 0 to (2^16)-1
- `::s32vector` signed exact integer in the range -(2^31) to (2^31)-1
- `::u32vector` unsigned exact integer in the range 0 to (2^32)-1
- `::s64vector` signed exact integer in the range -(2^63) to (2^63)-1
- `::u64vector` unsigned exact integer in the range 0 to (2^64)-1
- `f32vector` inexact small real
- `f64vector` inexact largest real

Each homogeneous vector datatype has an external representation which is supported by the read and write procedures and by the program parser. Each datatype also has a set of associated predefined procedures analogous to those available for Scheme's heterogeneous vectors.

As noted by Marc Feeley's specification, for each value of TAG in { s8, u8, s16, u16, s32, u32, s64, u64, f32,f64 }, if the datatype TAGvector is supported, then

- the external representation of instances of the datatype TAGvector is #TAG(...elements...).

  For example, `#u8(0 100 #xff)` is an u8vector of length 3 containing 0, 100 and 255; `#f64(-1.5)` is an f64vector of length 1 containing -1.5.

  Note that the syntax for float vectors conflicts with Standard Scheme which parses `#f32()` as 3 objects: `#f`, `32` and `()`. For this reason, conformance to this SRFI implies this minor nonconformance to Standard Scheme.

  This external representation is also available in program source code. For example, `(set! x '#u8(1 2 3))` will set x to the object `#u8(1 2 3)`. Literal homogeneous vectors must be quoted just like heterogeneous vectors must be. Homogeneous vectors can appear in quasiquotations but must not contain unquote or unquote-splicing forms (i.e. `` `(,x #u8(1 2)) `` is legal but `` `#u8(1 ,x 2) `` is not). This restriction is to accomomdate the many Scheme systems that use the read procedure to parse programs.
- the following predefined procedures are available:

`TAGvector?` *obj* [SRFI-4 procedure]
`make-TAGvector` *n* [*TAGvalue*] [SRFI-4 procedure]
`TAGvector` *TAGvalue ...* [SRFI-4 procedure]
`TAGvector-length` *TAGvect* [SRFI-4 procedure]
`TAGvector-ref` *TAGvect i* [SRFI-4 procedure]
`TAGvector-set!` *TAGvect i TAGvalue* [SRFI-4 procedure]
`TAGvector->list` *TAGvect* [SRFI-4 procedure]
`list->TAGvector` *TAGlist* [SRFI-4 procedure]

where obj is any Scheme object, n is a nonnegative exact integer, i is a nonnegative exact integer less than the length of the vector, TAGvect is an instance of the TAGvector datatype, TAGvalue is a number of the type acceptable for elements of the TAGvector datatype, and TAGlist is a proper list of numbers of the type acceptable for elements of the TAGvector datatype.

It is an error if TAGvalue is not the same type as the elements of the TAGvector datatype (for example if an exact integer is passed to f64vector). If the fill value is not specified, the content of the vector is unspecified but individual elements of the vector are guaranteed to be in the range of values permitted for that type of vector.

5.1.13 Control features

`procedure?` *obj* [procedure]

`apply` *proc arg1 ... args* [procedure]

`map` *proc list1 list2 ...* [library procedure]
`map!` *proc list1 list2 ...* [bigloo procedure]
`for-each` *proc list1 list2 ...* [library procedure]

`filter` *pred list ...* [library procedure]
`filter!` *pred list ...* [library procedure]

Strip out all elements of *list* for which the predicate *pred* is not true. The second version `filter!` is destructive:

```
(filter number? '(1 2 #\a "foo" foo 3)) ⇒ (1 2 3)
(let ((l (list 1 2 #\a "foo" 'foo 3)))
   (set! l (filter! number? l))
   l)                                   ⇒ (1 2 3)
```

`append-map` *proc list1 list2 ...* [library procedure]
`append-map!` *proc list1 list2 ...* [library procedure]

The expression

```
(append-map f clist1 clist2 ...)
```

is equivalent to:

```
(apply append (map f clist1 clist2 ...))
```

The expression

```
(append-map! f clist1 clist2 ...)
```

is equivalent to:

```
(apply append! (map f clist1 clist2 ...))
```

`filter-map` *pred list . . .* [bigloo procedure]

As `map` but only none `#f` values are accumulated in the resulting list. The Bigloo implementation complies with the SRFI-1 description.

```
(filter-map (lambda (x) (if (number? x) '- #f)) '(1 2 #\a "foo" foo 3)) ⇒ (- - -
)
```

`sort` *proc obj* [bigloo procedure]

`sort` *obj proc* [bigloo procedure]

Sorts *obj* according to *proc* test. The argument *obj* can either be a vector or a list. In either case, a copy of the argument is returned. For instance:

```
(let ((l '(("foo" 5) ("bar" 6) ("hux" 1) ("gee" 4))))
   (sort (lambda (x y) (string<? (car x) (car y))) l))
   ⇒ ((bar 6) (foo 5) (gee 4) (hux 1))
```

The second form (which uses *obj* before *proc* ensures backward compatibility with old Lisp systems, and older Bigloo versions. It is deprecated.

`force` *promise* [library procedure]

`call/cc` *proc* [bigloo procedure]

This function is the same as the `call-with-current-continuation` function of the R5RS, see Section "r5rs.info" in `R5RS`, but it is necessary to compile the module with the `-call/cc` option to use it, see Section See Chapter 31 [The Bigloo command line], page 275.

Note: Since `call/cc` is difficult to compile efficiently, one might consider using `bind-exit` instead. For this reason, we decided to enable `call/cc` only with a compiler option.

`bind-exit` *escape body* [bigloo syntax]

This form provides an escape operator facility. `bind-exit` evaluates the *body*, which may refer to the variable *escape* which will denote an "escape function" of one argument: when called, this escape function will return from the `bind-exit` form with the given argument as the value of the `bind-exit` form. The *escape* can only be used while in the dynamic extent of the form. Bindings introduced by `bind-exit` are immutable.

```
(bind-exit (exit)
 (for-each (lambda (x)
             (if (negative? x)
                 (exit x)))
           '(54 0 37 -3 245 19))
 #t)                                  ⇒ -3

(define list-length
  (lambda (obj)
    (bind-exit (return)
     (letrec ((r (lambda (obj)
                   (cond ((null? obj) 0)
                         ((pair? obj)
                          (+ (r (cdr obj)) 1))
                         (else (return #f))))))
          (r obj)))))

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

`unwind-protect` *expr protect* [bigloo syntax]

This form provides protections. Expression *expr* is evaluated. If this evaluation requires the invocation of an escape procedure (a procedure bounded by the `bind-exit` special form), *protect* is evaluated before the control jump to the exit procedure. If *expr* does not raise any exit procedure, `unwind-protect` has the same behaviour as the `begin` special form except that the value of the form is always the value of *expr*.

```
(define (my-open f)
   (if (file-exists? f)
       (let ((port (open-input-file f)))
          (if (input-port? port)
              (unwind-protect
                 (bar port)
                 (close-input-port port))))))
```

`dynamic-wind` *before thunk after* [procedure]

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

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

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

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

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

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

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
```

```
(dynamic-wind
  (lambda () (add 'connect))
  (lambda ()
    (add (call/cc
           (lambda (c0)
             (set! c c0)
             'talk1))))
  (lambda () (add 'disconnect)))
(if (< (length path) 4)
    (c 'talk2)
    (reverse path))))

⇒ (connect talk1 disconnect connect talk2 disconnect)
```

`unspecified` [bigloo procedure]

Returns the *unspecified* (noted as `#unspecified`) object with no specific property.

`try` *exp handler* [bigloo syntax]

This form is documented in Section Chapter 15 [Errors Assertions and Traces], page 173.

`values` *obj ...* [procedure]

Delivers all of its arguments to its continuation. Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value. `Values` might be defined as follows:

```
(define (values . things)
  (call/cc
    (lambda (cont) (apply cont things))))
```

`call-with-values` *producer consumer* [procedure]

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

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
    ⇒ 5
(call-with-values * -)
    ⇒ -1
```

`multiple-value-bind` (*var ...*) *producer exp ...* [bigloo syntax]
`receive` (*var ...*) *producer exp ...* [bigloo syntax]

Evaluates *exp ...* in a environment where *var ...* are bound from the evaluation of *producer*. The result of *producer* must be a call to `values` where the number of argument is the number of bound variables.

```
(define (bar a)
   (values (modulo a 5) (quotient a 5)))

(define (foo a)
   (multiple-value-bind (x y)
      (bar a)
      (print x " " y)))

(foo 354)
   ⊣ 4 70
```

5.2 Input and output

This section describes Scheme operation for reading and writing data. The section Section 5.7.2 [Files], page 79, describes functions for handling files.

5.2.1 Library functions

`call-with-input-file` *string proc* [library procedure]
`call-with-input-string` *string proc* [bigloo procedure]
`call-with-output-file` *string proc* [library procedure]
`call-with-append-file` *string proc* [library procedure]
`call-with-output-string` *proc* [library procedure]

These two procedures call *proc* with one argument, a port obtained by opening *string*. See Section "r5rs.info" in `R5RS`, for more details.

```
(call-with-input-file "/etc/passwd"
   (lambda (port)
      (let loop ((line (read-line port)))
         (if (not (eof-object? line))
             (begin
                (print line)
                (loop (read-line port)))))))
```

`input-port?` *obj* [procedure]
`input-string-port?` *obj* [procedure]
`output-port?` *obj* [procedure]
`output-string-port?` *obj* [procedure]
`port?` *obj* [procedure]

`input-port-name` *obj* [bigloo procedure]
`input-port-name-set!` *obj name* [bigloo procedure]
`output-port-name` *obj* [bigloo procedure]
`output-port-name-set!` *obj name* [bigloo procedure]

Returns/sets the file name for which *obj* has been opened.

`input-port-length` *obj* [bigloo (>=3.8d) procedure]

Returns the source number of bytes, i.e., the number characters contains in the source. Returns `-1` if that number is unknown (typically for a pipe).

`input-port-timeout-set!` *port time* [bigloo (>=2.8b) procedure]
`output-port-timeout-set!` *port time* [bigloo (>=2.8b) procedure]

These two functions limit the time an read or write operation may last. If the *time* limit (expressed in microseconds) exceeded, an exception of time `&io-timeout-error` is raised.

Setting a timeout equal to 0, restore the socket in blocking mode. Setting a timeout with a value lesser than 0 is ignored.

Note: ports created from sockets share their internal file descriptor. Hence it is erroneous to set a timeout for only one of the two ports. Both must be set.

`output-port-flush-hook` *port* [bigloo procedure]
`output-port-flush-hook-set!` *port hook* [bigloo procedure]

Returns (resp. sets) the *flush hook* of the output *port*. The flush hook is a procedure of two arguments, the output port and the number of characters that are to be actually

written out during the flush. It is unspecified when the hook is invoked, however, one may expect the C back-end to invoke the hook only when output buffers are full. The other back-ends (JVM and DOTNET) are likely to invoke the hook as soon as a character is to be written.

A flush hook can return two types of values:

- A string, which is then directly displayed to the system stream associated with the output port.
- An integer, which denotes the number of characters of the output port flush buffer (see `output-port-flush-buffer`) that have to be displayed on the system stream.

`output-port-flush-buffer` *port* [bigloo procedure]
`output-port-flush-buffer-set!` *port buffer* [bigloo procedure]

These functions gets and sets a buffer that can be used by program by the flush hooks. The runtime system makes no provision for automatically allocated these buffers that hence must be manually allocated by programs. The motivation for flush buffer is to allow programs to write flush hooks that don't have to allocate a new string each time invoked.

`output-port-close-hook` *port* [bigloo procedure]
`output-port-close-hook-set!` *port proc* [bigloo procedure]

Returns (resp. sets) the *close hook* of the output *port*. The close hook is a procedure of one argument, the closed port. The hook is invoked *after* the *port* is closed.

`input-port-close-hook` *port* [bigloo procedure]
`input-port-close-hook-set!` *port proc* [bigloo procedure]

Returns (resp. sets) the *close hook* of the input *port*. The close hook is a procedure of one argument, the closed port.

Example:

```
(let ((p (open-input-string "/etc/passwd")))
  (input-port-close-hook-set! p (lambda () (display 'done)))
  ...
  (close-input-port p))
```

`input-port-reopen!` *obj* [bigloo procedure]

Re-open the input port *obj*. That is, re-start reading from the first character of the input port.

`current-input-port` [procedure]
`current-output-port` [procedure]
`current-error-port` [bigloo procedure]

`with-input-from-file` *string thunk* [optional procedure]
`with-input-from-string` *string thunk* [optional procedure]
`with-input-from-procedure` *procedure thunk* [optional procedure]
`with-output-to-file` *string thunk* [optional procedure]
`with-append-to-file` *string thunk* [optional procedure]
`with-error-to-file` *string thunk* [bigloo procedure]
`with-output-to-string` *thunk* [bigloo procedure]

`with-output-to-procedure` *procedure thunk* [bigloo procedure]
`with-error-to-string` *thunk* [bigloo procedure]
`with-error-to-procedure` *procedure thunk* [bigloo procedure]

A port is opened from file *string*. This port is made the current input port (resp. the current output port or the current error port) and *thunk* is called. See Section "r5rs.info" in `R5RS`, for more details.

```
(with-input-from-file "/etc/passwd"
   (lambda ()
      (let loop ((line (read-line (current-input-port))))
         (if (not (eof-object? line))
             (begin
                (print line)
                (loop (read-line (current-input-port))))))))
```

`with-input-from-port` *port thunk* [bigloo procedure]
`with-output-to-port` *port thunk* [bigloo procedure]
`with-error-to-port` *port thunk* [bigloo procedure]

`with-input-from-port`, `with-output-to-port` and `with-error-to-port` all suppose *port* to be a legal port. They call *thunk* making *port* the current input (resp. output or error) port. None of these functions close *port* on the continuation of *thunk*.

```
(with-output-to-port (current-error-port)
   (lambda () (display "hello")))
```

`open-input-file` *file-name* [*buffer #f*] [*timeout 5000000*] [procedure]

If *file-name* is a regular file name, `open-input-file` behaves as the function defined in the Scheme report. If *file-name* starts with special prefixes it behaves differently. Here are the recognized prefixes:

- `|` (a string made of the characters `#\|` and `#\space`) Instead of opening a regular file, Bigloo opens an input pipe. The same syntax is used for output file.

  ```
  (define pin (open-input-file "| cat /etc/passwd"))
  (define pout (open-output-file "| wc -l"))

  (display (read pin) pout)
  (close-input-port pin)
  (newline pout)
  (close-output-port pout)
  ```

- `pipe:` Same as `|` .
- `file:` Opens a regular file.
- `gzip:` Opens a port on a gzipped filed. This is equivalent to `open-input-gzip-file`. Example:

  ```
  (with-input-from-file "gzip:bigloo.tar.gz"
     (lambda ()
        (send-chars (current-input-port) (current-output-port))))
  ```

- `string:` Opens a port on a string. This is equivalent to `open-input-string`. Example:

  ```
  (with-input-from-file "string:foo bar Gee"
     (lambda ()
        (print (read))
        (print (read))
        (print (read))))
  ```

```
⊣ foo
⊣ bar
⊣ Gee
```

- `http://server/path`

  Opens an *http* connection on `server` and open an input file on file `path`.

- `http://server:port-number/path`
- `http://user:password@server:port-number/path`

  Opens an *http* connection on `server`, on port number `port` with an authentication and open an input file on file `path`.

- `ftp://server/path`
- `ftp://user:password@server/path`

  Opens an *ftp* connection on `server` and open an input file on file `path`. Log in as anonymous.

- `ressource:`

  Opens a JVM *ressource* file. Opening a `ressource:` file in non JVM backend always return `#f`. On the JVM backend it returns a input port if the ressource exists. Otherwise, it returns `#f`.

The optional argument *buffer* can either be:

- A positive fixnum, this gives the size of the buffer.
- The boolean `#t`, a buffer is allocated.
- The boolean `#f`, the socket is unbufferized.
- A string, it is used as buffer.

The optional argument *timeout*, an integer represents a microseconds timeout for the open operation.

`open-input-gzip-file` *file-name* [*buffer #t*] [bigloo procedure]
`open-input-gzip-port` *input-port* [*buffer #t*] [bigloo procedure]

Open respectively a gzipped file for input and a port on a gzipped stream. Note that closing a gzip port opened from a port *pi* does not close the *pi* port.

```
(let ((p (open-input-gzip-file "bigloo.tar.gz")))
   (unwind-protect
      (read-line p1)
      (close-input-port p)))

(let* ((p1 (open-input-file "bigloo.tar.gz"))
       (p2 (open-input-gzip-port p1)))
   (unwind-protect
      (read-line p2)
      (close-input-port p2)
      (close-input-port p1)))
```

`open-input-zlib-file` *file-name* [*buffer #t*] [bigloo procedure]
`open-input-zlib-port` *input-port* [*buffer #t*] [bigloo procedure]

Open respectively a zlib file for input and a port on a zlib stream. Note that closing a zlib port opened from a port *pi* does not close the *pi* port.

`open-input-string` *string* [*start 0*] [*end*] [bigloo procedure]
`open-input-string!` *string* [*start 0*] [*end*] [bigloo procedure]

string must be a string, and *start* and *end* must be exact integers satisfying:

```
0 <= START <= END <= (string-length STRING)
```

The optional argument *end* defaults to `(string-length STRING)`.

Returns an `input-port` able to deliver characters from *string*.

The function `open-input-string!` acts as `open-input-string` but it might modify the string it receives as parameter.

`open-input-c-string` *string* [bigloo procedure]

Returns an `input-port` able to deliver characters from C *string*. The buffer used by the input port is the exact same string as the argument. That is, no buffer is allocated.

`open-input-ftp-file` *file-name* [*buffer #t*] [bigloo procedure]

Returns an `input-port` able to deliver characters from a remote file located on a FTP server.

Example:

```
(let ((p (open-input-ftp-file "ftp-sop.inria.fr/ls-lR.gz'')))
  (unwind-protect
     (read-string p)
     (close-input-port p)))
```

The file name may contain user authentication such as:

```
(let ((p (open-input-ftp-file "anonymous:foo@ftp-sop.inria.fr/ls-lR.gz'')))
  (unwind-protect
     (read-string p)
     (close-input-port p)))
```

`open-input-procedure` *procedure* [*buffer #t*] [bigloo procedure]

Returns an `input-port` able to deliver characters from *procedure*. Each time a character has to be read, the *procedure* is called. This procedure may returns a string of characters, or the boolean `#f`. This last value stands for the end of file.

Example:

```
(let ((p (open-input-procedure (let ((s #t))
  (lambda ()
     (if s
 (begin
                                          (set! s #f)
                                          "foobar")
 s))))))
   (read))
```

`unread-char!` *char* [*input-port*] [bigloo procedure]
`unread-string!` *string* [*input-port*] [bigloo procedure]
`unread-substring!` *string start end* [*input-port*] [bigloo procedure]

Pushes the given *char*, *string* or substring into the input-port. The next read character(s) will be the pushed ones. The *input-port* must be buffered and not be closed.

Example:

```
(define p (open-input-string "a ymbol c"))
```

```
(read p)                        ⇒ a
(read-char p)                   ⇒ #\space
(unread-char! #\s p)
(read p)                        ⇒ symbol
(read-char p)                   ⇒ #\space
(read p)                        ⇒ c
(char-ready? p)                 ⇒ #f
(unread-string! "sym1 sym2" p)
(char-ready? p)                 ⇒ #t
(read p)                        ⇒ sym1
(read p)                        ⇒ sym2
```

`open-output-file` *file-name* [procedure]
The same syntax as `open-input-file` for file names applies here. When a file name starts with '`| `', Bigloo opens an output pipe instead of a regular file.

`append-output-file` *file-name* [bigloo procedure]
If `file-name` exists, this function returns an `output-port` on it, without removing it. New output will be appended to *file-name*. If `file-name` does not exist, it is created.

`open-output-string` [bigloo procedure]
This function returns an *output string port*. This object has almost the same purpose as `output-port`. It can be used with all the printer functions which accept `output-port`. An output on a *output string port* memorizes all the characters written. An invocation of `flush-output-port` or `close-output-port` on an *output string port* returns a new string which contains all the characters accumulated in the port.

`get-output-string` *output-port* [bigloo procedure]
Given an output port created by `open-output-string`, returns a string consisting of the characters that have been output to the port so far.

`open-output-procedure` *proc* [*flush* [*close*]] [bigloo procedure]
This function returns an *output procedure port*. This object has almost the same purpose as `output-port`. It can be used with all the printer functions which accept `output-port`. An output on a *output procedure port* invokes the *proc* procedure each time it is used for writing. That is, *proc* is invoked with a string denoting the displayed characters. When the function `flush-output-port` is called on such a port, the optional *flush* procedure is invoked. When the function `close-output-port` is called on such a port, the optional *close* procedure is invoked.

`close-input-port` *input-port* [procedure]
`close-output-port` *output-port* [procedure]
According to R5RS, the value returned is unspecified. However, if *output-port* was created using `open-output-string`, the value returned is the string consisting of all characters sent to the port.

`closed-input-port?` *input-port* [procedure]
`closed-output-port?` *output-port* [procedure]
Predicates that return `#t` if and if their associated port is closed. Return `#f` otherwise.

`input-port-name` *input-port* [bigloo procedure]
Returns the name of the file used to open the *input-port*.

`input-port-position` *port* [bigloo procedure]
`output-port-position` *port* [bigloo procedure]
Returns the current position (a character number), in the *port*.

`set-input-port-position!` *port pos* [bigloo procedure]
`set-output-port-position!` *port pos* [bigloo procedure]
These functions set the file position indicator for *port*. The new position, measured in bytes, is specified by *pos*. It is an error to seek a port that cannot be changed (for instance, a procedure or a console port). The result of these functions is unspecified. An error is raised if the position cannot be changed.

`input-port-reopen!` *input-port* [bigloo procedure]
This function re-opens the input `input-port`. That is, it reset the position in the *input-port* to the first character.

`read` [*input-port*] [procedure]
`read/case` *case* [*input-port*] [bigloo procedure]
`read-case-sensitive` [*input-port*] [bigloo procedure]
`read-case-insensitive` [*input-port*] [bigloo procedure]
Read a lisp expression. The case sensitivity of `read` is unspecified. If have to to enforce a special behavior regarding the case, use `read/case`, `read-case-sensitive` or `read-case-insensitive`. Let us consider the following source code: The value of the `read/case`'s *case* argument may either be `upcase`, `downcase` or `sensitive`. Using any other value is an error.

```
(define (main argv)
   (let loop ((exp (read-case-sensitive)))
      (if (not (eof-object? exp))
          (begin
             (display "exp: ")
             (write exp)
             (display " [")
             (display exp)
             (display "]")
             (print " eq?: " (eq? exp 'FOO) " " (eq? exp 'foo))
             (loop (read-case-sensitive))))))
```

Thus:

```
> a.out
foo
  ⊣ exp: foo [foo] eq?: #f #t
FOO
  ⊣ exp: FOO [FOO] eq?: #t #f
```

`read/rp` *grammar port* [bigloo procedure]
`read/lalrp` *lalrg rg port* [*emptyp*] [bigloo procedure]
These functions are fully explained in Chapter 10 [Regular Parsing], page 129, and Chapter 11 [Lalr Parsing], page 137.

`define-reader-ctor` *symbol procedure* [bigloo procedure]
Note: This feature is experimental and might be removed in feature versions.

The present SRFI-10 (`http://srfi.schemers.org/srfi-10/srfi-10.html`) proposes an extensible external representation of Scheme values, a notational convention for future SRFIs. This SRFI adds `#,(` as a new token and extends production rules of the grammar for a Scheme reader. The `#,()` form can be used for example to denote values that do not have a convenient printed representation, as well for conditional code compilation. It is proposed that future SRFIs that contain new read syntax for values use the `#,()` notation with an appropriate tag symbol.

As a particular example and the reference implementation for the `#,()` convention, this SRFI describes an interpretation of the `#,()` external form as a read-time application.

Examples:

```
(define-reader-ctor 'list list)
(with-input-from-string "#,(list 1 2 #f \"4 5\")" read) ⇒ (1 2 #f "4 5")

(define-reader-ctor '+ +)
(with-input-from-string "#,(+ 1 2)" read) ⇒ 3
```

`set-read-syntax!` *char procedure* [bigloo procedure]

Note: This feature is experimental and might be removed in feature versions.

Registers a function *procedure* to be invoked with one argument, an input-port, that is invoked when the reader hits an unparsed character.

Example:

```
(set-read-syntax! #\{
   (lambda (port)
      (let loop ((c (peek-char port)) (exps '()))
 (cond ((eof-object? c)
(error "{" "EOF encountered while parsing { ... } clause" port))
       ((char=? c #\})
(read-char port)   ; discard
`(begin ,@(reverse exps)))
       ((char-whitespace? c)
(read-char port)   ; discard whitespace
(loop (peek-char port) exps))
       (else
(let ((exp (read port)))
   (loop (peek-char port)
                     (cons exp exps))))))))
```

`read-char` [*port*] [procedure]
`read-byte` [*port*] [procedure]
`peek-char` [*port*] [procedure]
`peek-byte` [*port*] [procedure]
`eof-object?` *obj* [procedure]

`char-ready?` [*port*] [procedure]

As specified in the R5Rs, Section "r5rs.info" in `R5RS`, `char-ready?` returns `#t` if a character is ready on the input *port* and returns `#f` otherwise. If '`char-ready`' returns `#t` then the next '`read-char`' operation on the given *port* is guaranteed not to hang. If the *port* is at end of file then '`char-ready?`' returns `#t`. *Port* may be omitted, in which case it defaults to the value returned by '`current-input-port`'.

When using `char-ready?` consider the latency that may exists before characters are available. For instance, executing the following source code:

```
(let* ((proc (run-process "/bin/ls" "-l" "/bin" output: pipe:))
       (port (process-output-port proc)))
   (let loop ((line (read-line port)))
      (print "char ready " (char-ready? port))
      (if (eof-object? line)
          (close-input-port port)
          (begin
             (print line)
             (loop (read-line port))))))
```

Produces outputs such as:

```
char ready #f
total 7168
char ready #f
-rwxr-xr-x    1 root     root         2896 Sep  6  2001 arch
char ready #f
-rwxr-xr-x    1 root     root        66428 Aug 25  2001 ash
char ready #t
...
```

For a discussion of Bigloo processes, see Section 5.7.3 [Process], page 83.

Note: Thanks to Todd Dukes for the example and the suggestion of including it this documentation.

`read-line` [*input-port*] [bigloo procedure]
`read-line-newline` [*input-port*] [bigloo procedure]

Reads characters from *input-port* until a `#\Newline`, a `#\Return` or an `end of file` condition is encountered. `read-line` returns a newly allocated string composed of the characters read.

The strings returned by `read-line` do not contain the newline delimiters. The strings returned by `read-line-newline` do contain them.

`read-lines` [*input-port*] [bigloo procedure]

Accumulates all the line of an *input-port* into a list.

`read-of-strings` [*input-port*] [bigloo procedure]

Reads a sequence of non-space characters on *input-port*, makes a string of them and returns the string.

`read-string` [*input-port*] [bigloo procedure]

Reads all the characters of *input-port* into a string.

`read-chars` *size* [*input-port*] [bigloo procedure]
`read-chars!` *buf size* [*input-port*] [bigloo procedure]

The function `read-chars` returns a newly allocated strings made of *size* characters read from *input-port* (or from `(current-input-port)` if *input-port* is not provided). If less than *size* characters are available on the input port, the returned string is smaller than *size*. Its size is the number of available characters.

The function `read-chars!` fills the buffer *buf* with at most *size* characters.

`read-fill-string!` *s o len* [*input-port*] [bigloo procedure]

Fills the string *s* starting at offset *o* with at most *len* characters read from the input port *input-port* (or from `(current-input-port)` if *input-port* is not provided). This function returns the number of read characters (which may be smaller than *len* if less characters are available) or the end of file object. The argument `len` is a small integer.

The function `read-fill-string!` is similar to `read-chars!` except that it returns the *end-of-file* object on termination while `read-chars!` returns 0.

Example:

```
(let ((s (make-string 10 #\-)))
   (with-input-from-string "abcdefghijlkmnops"
      (lambda ()
         (read-fill-string! s 3 5)
         s)))
   ⇒ ---abcde--
```

`port->string-list` *input-port* [bigloo procedure]

Returns a list of strings composed of the elements of *input-port*.

`port->list` *input-port reader* [bigloo procedure]
`port->sexp-list` *input-port* [bigloo procedure]

`Port->list` applies reader to port repeatedly until it returns EOF, then returns a list of results. `Port->list-sexp` is equivalent to `(port->list read port)`.

`file->string` *path* [bigloo procedure]

This function builds a new string out of all the characters of the file *path*. If the file cannot be open or read, an `IO_EXCEPTION` is raised.

`send-chars` *input-port output-port* [*len*] [*offset*] [bigloo procedure]
`send-file` *filename output-port* [*len*] [*offset*] [bigloo procedure]

Transfer the characters from *input-port* to *output-port*. This procedure is sometimes mapped to a system call (such as `sendfile` under Linux) and might thus be more efficient than copying the ports by hand. The optional argument *offset* specifies an offset from which characters of *input-port* are sent. The function `send-chars` returns the number of characters sent.

The function `send-file` opens the file *filename* in order to get its input port. On some backends, `send-file` might be more efficient than `send-chars` because it may avoid creating a full-fledged Bigloo `input-port`.

Note that the type of *len* and *offset* is `elong` (i.e., exact long), which is also returned by `file-size`.

`write` *obj* [*output-port*] [library procedure]
`display` *obj* [*output-port*] [library procedure]
`print` *obj . . .* [bigloo procedure]

This procedure allows several objects to be displayed. When all these objects have been printed, `print` adds a newline.

`display*` *obj . . .* [bigloo procedure]

This function is similar to `print` but does not add a newline.

`fprint` *output-port obj . . .* [bigloo procedure]
This function is the same as `print` except that a port is provided.

`write-char` *char* [*output-port*] [procedure]
`write-byte` *byte* [*output-port*] [procedure]
These procedures write a char (respec. a byte, i.e., in integer in the range 0..255) to the *output-port*.

`newline` [*output-port*] [procedure]
`flush-output-port` *output-port* [bigloo procedure]
This procedure flushes the output port *output-port*. This function *does not* reset characters accumulated in string port. For this uses, `reset-output-port`.

`newline` [*output-port*] [procedure]
`reset-output-port` *output-port* [bigloo procedure]
This function is equivalent to `flush-output-port` but in addition, for string ports, it reset the internal buffer that accumulates the displayed characters.

`format` *format-string* [*objs*] [bigloo procedure]
Note: Many thanks to Scott G. Miller who is the author of SRFI-28. Most of the documentation of this function is copied from the SRFI documentation.

Accepts a message template (a Scheme String), and processes it, replacing any escape sequences in order with one or more characters, the characters themselves dependent on the semantics of the escape sequence encountered.

An escape sequence is a two character sequence in the string where the first character is a tilde `~`. Each escape code's meaning is as follows:

- `~a` The corresponding value is inserted into the string as if printed with display.
- `~s` The corresponding value is inserted into the string as if printed with write.
- `~%` or `~n` A newline is inserted A newline is inserted.
- `~~` A tilde `~` is inserted.
- `~r` A return (`#\Return`) is inserted.
- `~v` The corresponding value is inserted into the string as if printed with display followed by a newline. This tag is hence equivalent to the sequence `~a~n`.
- `~c` The corresponding value must be a character and is inserted into the string as if printed with write-char.
- `~d`, `~x`, `~o`, `~b` The corresponding value must must be a number and is printed with radix 16, 8 or 2.
- `~l` If the corresponding value is a proper list, its items are inserted into the string, separated by whitespaces, without the surrounding parenthesis. If the corresponding value is not a list, it behaves as `~s`.
- `~(<sep>)` If the corresponding value is a proper list, its items are inserted into the string, separated from each other by *sep*, without the surrounding parenthesis. If the corresponding value is not a list, it behaves as `~s`.
- `~Ndxob` Print a number in N columns with space padding.
- `~N,<padding>dxob` Print a number in *num* columns with *padding* padding.

`~a` and `~s`, when encountered, require a corresponding Scheme value to be present after the format string. The values provided as operands are used by the escape sequences in order. It is an error if fewer values are provided than escape sequences that require them.

`~%` and `~~` require no corresponding value.

```
(format "Hello, ~a" "World!")
   ⊣ Hello, World!
(format "Error, list is too short: ~s~%" '(one "two" 3))
   ⊣ Error, list is too short: (one "two" 3)
(format "a ~l: ~l" "list" '(1 2 3))
   ⊣ a list: 1 2 3
(format "a ~l: ~(, )" "list" '(1 2 3))
   ⊣ a list: 1, 2, 3
(format "~3d" 4)
   ⊣   4
(format "~3,-d" 4)
   ⊣ --4
(format "~3x" 16)
   ⊣  10
(format "~3,0d" 5)
   ⊣ 005
```

`printf` *format-string* [*objs*] [bigloo procedure]
`fprintf` *port format-string* [*objs*] [bigloo procedure]

Formats *objs* to the current output port or to the specified *port*.

`pp` *obj* [*output-port*] [bigloo procedure]

Pretty print *obj* on *output-port*.

`*pp-case*` [bigloo variable]

Sets the variable to `respect`, `lower` or `upper` to change the case for pretty-printing.

`*pp-width*` [bigloo variable]

The width of the pretty-print.

`write-circle` *obj* [*output-port*] [bigloo procedure]

Display recursive object *obj* on *output-port*. Each component of the object is displayed using the `write` library function.

`display-circle` *obj* [*output-port*] [bigloo procedure]

Display recursive object *obj* on *output-port*. Each component of the object is displayed using the `display` library function.

For instance:

```
(define l (list 1 2 3))
(set-car! (cdr l) l)
(set-car! (cddr l) l)
(display-circle l)  ⊣ #0=(1 #0# #0#)
```

`display-string` *string output-port* [bigloo procedure]
`display-substring` *string start end output-port* [bigloo procedure]

String must be a string, and *start* and *end* must be exact integers satisfying `0 <= start <= end <= (string-length string)`.

`Display-substring` displays a string formed from the characters of string beginning with index *start* (inclusive) and ending with index *end* (exclusive).

`password` [*prompt*] [bigloo procedure]

Reads a password from the current input port. The reading stops when the user hits the ,(code "Enter") key.

`open-pipes` [*name*] [bigloo procedure]

Opens a bi-directional pipes. Returns two values, an `input-port` and an `output-port`. The optional argument *name* is only used for debugging.

Example:

```
(multiple-value-bind (in out)
  (open-pipes "my pipe")
  (write-char #\z out)
  (flush-output-port out))
```

`select` [*:timeout 0*] [*:read '()*] [*:write '()*] [*:except '()*] [bigloo procedure]

A wrapper of the Posix `select` function. Returns three values, the three lists of objects that are ready for reading, respectively writing, or that are in error.

Example:

```
(define *inpipe* #f)
(define *outpipe* #f)
(define *watch-mutex* (make-mutex "watch"))
(define *sockets* '())

(define (watch socket onclose)
   (synchronize *watch-mutex*
      (set! *sockets* (cons socket *sockets*))
      (if *outpipe*
  (begin
     (write-char *outpipe*)
     (flush-output-port *outpipe*))
  (thread-start!
     (instantiate::hopthread
(body (watch-thread onclose)))))))

(define (watch-thread onclose)
   (let loop ()
      (synchronize *watch-mutex*
 (unless *inpipe*
    (multiple-value-bind (in out)
       (open-pipes)
       (set! *inpipe* in)
       (set! *outpipe* out))))
      (multiple-value-bind (readfs _ _)
 (select :read (cons *inpipe* *sockets*))
 (let ((socks (filter socket? readfs)))
    (for-each onclose socks)
    (synchronize *watch-mutex*
       (for-each (lambda (s)
    (set! *sockets* (remq! s *sockets*)))
  socks)
       (unless (pair? *sockets*)
  (close-input-port *inpipe*)
  (close-output-port *outpipe*)
  (set! *inpipe* #f)
  (set! *outpipe* #f)))
    (when *outpipe*
       (loop))))))
```

`lockf` *output-port command* [*len 0*] [bigloo procedure]
Lock a file descriptor or an output port. It is an error to call `lockf` with an port not open on a plain file (i.e., a port open with `open-output-file`, or its variants).

The *command* argument is one of:

- `lock`: locks the file, raises an error on failure.
- `ulock`: unlocks the file, raises an error on failure.
- `test`: tests whether a file is locked or not.
- `tlock`: tries to lock a file, return `#t` upon success and `#f` otherwise.

The argument *len* is the portion of the file to be locked.

5.2.2 mmap

The `mmap` function asks to map a file into memory. This memory area can be randomly accessed as a string. In general using `mmap` improves performance in comparison with equivalent code using regular ports.

`mmap?` *obj* [bigloo procedure]
Returns `#t` if and only if *obj* has been produced by `open-mmap`. Otherwise, it returns `#f`.

`open-mmap` *path* [*mode*] [bigloo procedure]
Maps a file *path* into memory. The optional argument *mode* specifies how the file is open. The argument can be:

- `read: #t` The memory can be read
- `read: #f` The memory cannot be read
- `write: #t` The memory can be written
- `write: #f` The memory is read-only.

`string->mmap` *string* [*mode*] [bigloo procedure]
Wrap a Bigloo string into a mmap object.

`mmap-name` *mm* [bigloo procedure]
Returns the file name of the memory map *mm*.

`close-mmap` *mm* [bigloo procedure]
Closes the memory mapped. Returns `#t` on success, `#f` otherwise.

`mmap-length` *mm* [bigloo procedure]
Returns the length, an exact integer, of the memory mapped.

`mmap-read-position` *mm* [bigloo procedure]
`mmap-read-position-set!` *mm offset* [bigloo procedure]
`mmap-write-position` *mm* [bigloo procedure]
`mmap-write-position-set!` *mm offset* [bigloo procedure]
Returns and sets the read and write position of a memory mapped memory. The result and the argument are exact integers.

`mmap-ref` *mm offset* [bigloo procedure]
Reads the character in *mm* at *offset*, an exact long (::elong). This function sets the read position to `offset + 1`.

`mmap-set!` *mm offset char* [bigloo procedure]
Writes the character *char* in *mm* at *offset*, an exact long (::elong). This function sets the write position to `offset + 1`.

`mmap-substring` *mm start end* [bigloo procedure]
Returns a newly allocated string made of the characters read from *mm* starting at position *start* and ending at position *end - 1*. If the values `start` and `end` are not ranged in `[0...(mmap-length mm)]`, an error is signaled. The function `mmap-substring` sets the read position to `end`.

`mmap-substring-set!` *mm start str* [bigloo procedure]
Writes the string *str* to *mm* at position *start*. If the values `start` and `start + (string-length str)` are not ranged in `[0...(mmap-length mm)[`, an error is signaled. The function `mmap-substring` sets the write position to `start + (string-length str)`.

`mmap-get-char` *mm* [bigloo procedure]
`mmap-put-char!` *mm c* [bigloo procedure]
`mmap-get-string` *mm len* [bigloo procedure]
`mmap-put-string!` *mm str* [bigloo procedure]
These functions get (resp. put) character and strings into a memory mapped area. They increment the read (resp. write) position. An error is signaled if the characters read (resp. writen) outbound the length of the memory mapped.

5.2.3 Zip

`port->gzip-port` *input-port* [*buffer #t*] [bigloo procedure]
`port->zlib-port` *input-port* [*buffer #t*] [bigloo procedure]
`port->inflate-port` *input-port* [*buffer #t*] [bigloo procedure]
These functions take a regular port as input (*input-port*). They construct a new port that automatically *unzip* the read characters. The `inflate` version does not parse a gunzip-header before inflating the content.

`open-input-inflate-file` *path* [*buffer #t*] [bigloo procedure]
These function open a gzipped file for input. The file is automatically unzipped when the characters are read. It is equivalent to:

```
(let ((p (open-input-port path)))
  (port->gzip-port p))
```

The function `open-input-inflate-file` is similar to `open-input-gzip-file` but it does not parse a gunzip-header before inflating the content.

`gunzip-sendchars` *input-port output-port* [bigloo procedure]
`inflate-sendchars` *input-port output-port* [bigloo procedure]
Transmit all the characters from the gzipped *input-port* to the *output-port*.

Note that the function `send-chars` can also be used on gzipped input-ports.

`gunzip-parse-header` *input-port* [bigloo procedure]
Parse the header of *input-port*. Returns `#f` if and only if the port is not gzipped.

5.2.4 Tar

`tar-read-header` [*input-port*] [bigloo procedure]
Reads a tar header from *input-port*. If the input-port does not conform the tar format, an IO exception is raised. On success a *tar-header* descriptor is returned.

`tar-read-block` *tar-header* [*input-port*] [bigloo procedure]
Reads the content of the *tar-header* block.

`tar-round-up-to-record-size` *int* [bigloo procedure]
Rounds up tar-block sizes.

`tar-header-name` *tar-header* [bigloo procedure]
`tar-header-mode` *tar-header* [bigloo procedure]
`tar-header-uid` *tar-header* [bigloo procedure]
`tar-header-gid` *tar-header* [bigloo procedure]
`tar-header-size` *tar-header* [bigloo procedure]
`tar-header-mtim` *tar-header* [bigloo procedure]
`tar-header-checksum` *tar-header* [bigloo procedure]
`tar-header-type` *tar-header* [bigloo procedure]
`tar-header-linkname` *tar-header* [bigloo procedure]
`tar-header-uname` *tar-header* [bigloo procedure]
`tar-header-gname` *tar-header* [bigloo procedure]
`tar-header-devmajor` *tar-header* [bigloo procedure]
`tar-header-devminir` *tar-header* [bigloo procedure]
Return various information about *tar-header*.

The following example simulates the Unix command `tar xvfz`:

```
(define (untar path)
   (let ((pz (open-input-gzip-port path)))
      (unwind-protect
 (let loop ((lst '()))
    (let ((h (tar-read-header pz)))
       (if (not h)
  lst
  (case (tar-header-type h)
     ((dir)
      (let ((path (tar-header-name h)))
 (if (make-directory path)
     (loop lst)
     (error 'untar
    "Cannot create directory"
    path))))
      ((normal)
       (let* ((path (tar-header-name h))
      (dir (dirname path)))
  (when (and (file-exists? dir) (not (directory? dir)))
     (delete-file dir))
  (unless (file-exists? dir)
     (make-directory dir))
```

```
  (with-output-to-file path
     (lambda ()
(display (tar-read-block h pz))))
  (loop (cons path lst))))
      (else
       (error 'untar
      (format "Illegal file type '~a'"
      (tar-header-type h))
      (tar-header-name h)))))))
 (close-input-port pz))))
```

`untar` *input-port* [*:directory* (*pwd*)] [*:file #f*] [bigloo procedure]

Untars the archive whose content is provided by the input port *input-port*.

- If *:file* is provided, `untar` extract the content of the file named *:file* and returns a string. The file name must exactly matches the files of the archive files names. If the file does not exist, `untar` returns `#f`.
- If *:file* is not provided, it *untars* the whole content, in the directory denoted by *:directory*, which defaults to `(pwd)`. The function `untar`, returns the whole list of created directories and files.

5.3 Serialization

`string->obj` *string #!optional extension* [bigloo procedure]

This function converts a *string* which has been produced by `obj->string` into a Bigloo object.

New in Bigloo 4.2a: The *extension* parameter is used to decode *extension* sequences. Theses sequences of characters are introduced by the `X` character. To decode an extension, the unserializer starts decoding the item following the `X` as a regular serialized item. Then, if the *extension* parameter is bound to a function, the unserializer calls that function and use the returned value as the unserialized object. If the *extension* argument is not a function, the unserializer return the ream item.

`obj->string` *object* [bigloo procedure]

This function converts into a string *any* Bigloo *object* which does not contain a procedure.

The implementation of the last two functions ensures that for every Bigloo object `obj` (containing no procedure), the expression:

```
(equal? obj (string->obj (obj->string obj)))
   ⇒ #t
```

`binary-port?` *obj* [bigloo procedure]
`open-output-binary-file` *file-name* [bigloo procedure]
`append-output-binary-file` *file-name* [bigloo procedure]
`open-input-binary-file` *file-name* [bigloo procedure]
`close-binary-port` *binary-port* [bigloo procedure]
`flush-binary-port` *binary-port* [bigloo procedure]
`input-obj` *binary-port* [bigloo procedure]

`output-obj` *binary-port obj* [bigloo procedure]

Bigloo allows Scheme objects to be dumped into, and restored from, files. These operations are performed by the previous functions. The dump and the restore use the two functions `obj->string` and `string->obj`.

It is also possible to use a binary file as a flat character file. This can be done by the means of `output-char`, `input-char`, `output-string`, and `input-string` functions.

`input-char` *binary-port* [bigloo procedure]
`output-char` *binary-port char* [bigloo procedure]
`output-byte` *binary-port byte* [bigloo procedure]

The function `input-char` reads a single character from a *binary-port*. It returns the read character or the `end-of-file` object. The function `output-char` and `output-byte` writes a character, respectively a byte, into a *binary-port*.

`input-string` *binary-port len* [bigloo procedure]
`output-string` *binary-port* [bigloo procedure]

The function `input-string` reads a string from a *binary-port* of maximum length *len*. It returns a newly allocated string whose length is possibly smaller than *len*. The function `output-string` writes a string into a *binary-port*.

`input-fill-string!` *binary-port string* [bigloo procedure]

Fills a string with characters read from *binary-port* with at most the length of *string*. The function returns the number of filled characters.

`register-procedure-serialization!` *serializer unserializer* [bigloo procedure]
`register-custom-serialization!` *ident serializer unserializer* [bigloo procedure]
`register-process-serialization!` *serializer unserializer* [bigloo procedure]
`register-opaque-serialization!` *serializer unserializer* [bigloo procedure]

There is no existing portable method to dump and restore a procedure. Thus, if `obj->string` is passed a procedure, it will emit an error message. Sometime, using strict restrictions, it may be convenient to use an ad-hoc framework to serialize and unserialize procedures. User may specify there own procedure serializer and unserializer. This is the role of `register-procedure-serialization!`. The argument *serializer* is a procedure of one argument, converting a procedure into a characters strings. The argument *unserializer* is a procedure of one argument, converting a characters string into a procedure. It belongs to the user to provide correct serializer and unserializer.

Here is an example of procedure serializer and unserializer that may be correct under some Unix platform:

```
(module foo
   (extern (macro %sprintf::int (::string ::string ::procedure) "sprintf")))

(define (string->procedure str)
   (pragma "(obj_t)(strtoul(BSTRING_TO_STRING($1), 0, 16))" str))

(define (procedure->string proc)
   (let ((item (make-string 10)))
      (%sprintf item "#p%lx" proc)
```

```
      item))

(register-procedure-serialization! procedure->string string->procedure)

(let ((x 4))
   (let ((obj (cons "toto" (lambda (y) (+ x y)))))
      (let ((nobj (string->obj (obj->string obj))))
 (print ((cdr nobj) 5)))))
```

`register-class-serialization!` *class serializer unserializer* [bigloo procedure]

Register a serializer/unserializer for a class. Subclasses of *class* inherit this serializer.

```
(module class-serialization-example
   (static (class point::object (x (default 10)) (y (default 20)))))

(register-class-serialization! point
       (lambda (o)
  (with-access::point o (x y)
     (cons x y)))
       (lambda (l)
  (instantiate::point
     (x (car l))
     (y (cdr l)))))

(let ((o (instantiate::point)))
   (let ((s (obj->string (list o o))))
      (print (string-for-read s))
      (let ((l (string->obj s)))
 (print l)
 (eq? (car l) (cadr l))))) ⇒ #t
```

`get-procedure-serialization` [bigloo procedure]
`get-custom-serialization` *ident* [bigloo procedure]
`get-process-serialization` [bigloo procedure]
`get-opaque-serialization` [bigloo procedure]
`get-class-serialization` *class* [bigloo procedure]

Returns the a multiple-values whose first element is the current procedure serializer and whose second element is the current procedure unserializer. If no serializer/unserializer is defined, these procedures return the values `#f #f`.

5.4 Bit manipulation

These procedures allow the manipulation of fixnums as bit-fields.

`bit-or` *i1 i2* [bigloo procedure]
`bit-orelong` *i1 i2* [bigloo procedure]
`bit-orllong` *i1 i2* [bigloo procedure]
`bit-xor` *i1 i2* [bigloo procedure]
`bit-xorelong` *i1 i2* [bigloo procedure]
`bit-xorllong` *i1 i2* [bigloo procedure]
`bit-and` *i1 i2* [bigloo procedure]
`bit-andelong` *i1 i2* [bigloo procedure]
`bit-andllong` *i1 i2* [bigloo procedure]
`bit-not` *i* [bigloo procedure]

`bit-notelong` *i* [bigloo procedure]
`bit-notllong` *i* [bigloo procedure]
`bit-lsh` *i1 i2* [bigloo procedure]
`bit-lshelong` *i1 i2* [bigloo procedure]
`bit-lshllong` *i1 i2* [bigloo procedure]
`bit-rsh` *i1 i2* [bigloo procedure]
`bit-ursh` *i1 i2* [bigloo procedure]
`bit-rshelong` *i1 i2* [bigloo procedure]
`bit-rshllong` *i1 i2* [bigloo procedure]
`bit-urshelong` *i1 i2* [bigloo procedure]
`bit-urshllong` *i1 i2* [bigloo procedure]

```
(bit-or 5 3)                    ⇒ 7
(bit-orelong #e5 #e3)           ⇒ #e7
(bit-xor 5 3)                   ⇒ 6
(bit-andllong #l5 #l3)          ⇒ #l1
(bit-not 5)                     ⇒ -6
(bit-lsh 5 3)                   ⇒ 40
(bit-rsh 5 1)                   ⇒ 2
```

5.5 Weak Pointers

Bigloo may support weak pointers. In order to activate this support, Bigloo must be configured with the `finalization` enabled. That is, the `configure` script must be invoked with the option `--finalization=yes`. When the finalization and weak pointers support is enabled, Bigloo defines the `cond-expand` properties `bigloo-finalizer` and `bigloo-weakptr`. Then a program may test the support with expressions such as:

```
(cond-expand
  (bigloo-weakptr <something>)
  (else <something-else>))
```

Weak pointers are pointers to objects which can be collected by the garbage collector if they are weakly pointed to. An object is weakly pointed to if the only pointers to it are weak pointers. Weakly pointed objects can be collected by the garbage collector, and all the weak pointers to such objects will cease to point to it and point to `#unspecified` instead.

`make-weakptr` *data* [bigloo procedure]
Creates a weak pointer to *data*.

`weakptr?` *obj* [bigloo procedure]
Returns `#t` if *obj* is a weak pointer, constructed by `make-weakptr`.

`weakptr-data` *ptr* [bigloo procedure]
Returns the data object pointed to by *ptr*. If the object has been collected, it returns `#unspecified`.

5.6 Hash Tables

Bigloo offers hash tables with support for weak pointers. Here are described functions which define and use them.

`make-hashtable` [*bucket-len*] [*max-bucket-len*] [*eqtest*] [*hash*] [*weak-keys*] [*weak-data*] [bigloo procedure]

`create-hashtable` [*:size*] [*:max-bucket-len*] [*:eqtest*] [*:hash*] [*:weak*] [*:max-length*] [*:bucket-expansion*] [bigloo procedure]

Defines an hash table for which the number of buckets is *size*. The variable *max-bucket-len* specify when the table should be resized. If provided, these two values have to be exact integers greater or equal to 1. Normally you could ignore *size* and *max-bucket-len* arguments and call `make-hashtable` with no argument at all. The argument *eqtest* enables the specification of a comparison function. The first argument of this function is the keys contained in the table. The second argument is the searched key. By default hash tables rely on `hashtable-equal?`, which is defined as:

```
(define (hashtable-equal? obj1 obj2)
   (or (eq? obj1 obj2)
       (and (string? obj1)
            (string? obj2)
            (string=? obj1 obj2))))
```

The argument *hash* specifies an hashing function. It defaults to *get-hashnumber*. The arguments *weak-keys*, *weak-data*, and *weak-both* specify respectively whether the hash table should use weak pointers to store the keys and/or the data. By default a hash table uses strong pointers for both keys and data. Each optional arguments *size*, *max-bucket-len*, *eqtest*, *hash*, *weak-keys*, and *weak-data* can be bound to the Bigloo value `#unspecified` which forces its default.

The argument *max-length* specifies a maximum length (in number of buckets) for this hashtable. It defaults to `16384`. If during the execution, the hashtable tries to expand itself more than *max-length*, an exception is raised. This feature helps debugging incorrect hashtable uses because excessive expansion is generally the signs of an incorrect behavior. Excessive expansions, cause the garbage collector to crash at some point. This debugging feature can be disabled by specifying a negative max length, in which case, no check is performed at runtime.

The argument *bucket-expansion* controls how *max-bucket-len* is expanded each time the table grows. This is a floating point number that is a multiplicative coefficient. It defaults to `1.2`.

The function `create-hashtable` is equivalent to `make-hashtable` but it uses a keyword interface. The keyword argument `weak` can either be `none`, `data`, or `keys`.

`hashtable?` *obj* [bigloo procedure]

Returns `#t` if *obj* is an hash table, constructed by `make-hashtable`.

`hashtable-weak-keys?` *table* [bigloo procedure]

Returns `#t` if *table* is a hash table with weakly pointed keys.

`hashtable-weak-data?` *table* [bigloo procedure]

Returns `#t` if *table* is a hash table with weakly pointed data.

`hashtable-size` *table* [bigloo procedure]

Returns the number of entries contained in *table*. Note that for a weak hash table the size does not guarantee the real size, since keys and/or data can dissapear before the next call to the hash table.

`hashtable-contains?` *table key* [bigloo procedure]
Returns the boolean `#t` if it exists at least one entry whose key is *key* in *table*. If not entry is found `#f` is returned. Note that for a weak hash table, the fact this procedure returns `#t` does not guarantee that the key (or its associated data) will not dissapear before the next call to the hash table.

`hashtable-get` *table key* [bigloo procedure]
Returns the entry whose key is *key* in *table*. If no entry is found, or if the key and/or value is weakly pointed to and has dissapeard, `#f` is returned.

`hashtable-put!` *table key obj* [bigloo procedure]
Puts *obj* in *table* under the key *key*. This function returns the object bound in the table. If there was an object *obj-old* already in the table with the same key as *obj*, this function returns *obj-old*; otherwise it returns *obj*.

`hashtable-remove!` *table key* [bigloo procedure]
Removes the object associated to *key* from *table*, returning `#t` if such object was bound in table and `#f` otherwise.

`hashtable-add!` *table key update-fun obj init-value* [bigloo procedure]
If key is already in table, the new value is calculated by `(update-fun obj current-value)`. Otherwise the *table* is extended by an entry linking key and `(update-fun obj init-value)`.

`hashtable-update!` *table key update-fun init-value* [deprecated bigloo procedure]
If key is already in table, the new value is calculated by `(update-fun current-value)`. Otherwise the *table* is extended by an entry linking key and `init-value`.

`hashtable->vector` *table* [bigloo procedure]
`hashtable->list` *table* [bigloo procedure]
Returns the hash table *table*'s data as a vector (respectively a list). If the hash table is weak, the result will consist only of the data which haven't dissapeared yet and whose keys haven't dissapeared either.

`hashtable-key-list` *table* [bigloo procedure]
Returns the list of keys used in the *table*. If the hash table is weak, the result will consist only of the keys which haven't dissapeared yet and whose data haven't dissapeared either.

`hashtable-map` *table fun* [bigloo procedure]
Returns a list whose elements are the result of applying *fun* to each of the keys and elements of *table* (no order is specified). In consequence, *fun* must be a procedure of two arguments. The first one is a key and the second one, an associated object. If the hash table is weak, *fun* will only be mapped on sets of key/datum which haven't dissapeared yet.

`hashtable-for-each` *table fun* [bigloo procedure]
Applies *fun* to each of the keys and elements of *table* (no order is specified). In consequence, *fun* must be a procedure of two arguments. The first one is a key and the second one, an associated object. If the hash table is weak, *fun* will only be called on sets of key/datum which haven't dissapeared yet.

`hashtable-filter!` *table fun* [bigloo procedure]
Filter out elements from *table* according to predicate *fun*. If the hash table is weak, *fun* will only be called on sets of key/datum which haven't dissapeared yet.

`hashtable-clear!` *table* [bigloo procedure]
Remove all the elements from *table*.

Here is an example of hash table.

```
(define *table* (make-hashtable))

(hashtable-put! *table* "toto" "tutu")
(hashtable-put! *table* "tata" "titi")
(hashtable-put! *table* "titi" 5)
(hashtable-put! *table* "tutu" 'tutu)
(hashtable-put! *table* 'foo 'foo)

(print (hashtable-get *table* "toto"))
   ⊣ "tutu"
(print (hashtable-get *table* 'foo))
   ⊣ 'foo
(print (hashtable-get *table* 'bar))
   ⊣ #f

(hashtable-for-each *table* (lambda (key obj) (print (cons key obj))))
   ⊣ ("toto" . "tutu")
     ("tata" . "titi")
     ("titi" . 5)
     ("tutu" . TUTU)
     (foo . foo)
```

`object-hashnumber` *object* [bigloo generic]
This generic function computes a hash number of the instance *object*.

Example:

```
(define-method (object-hashnumber pt::point)
   (with-access::point pt (x y)
      (+fx (*fx x 10) y)))
```

`string-hash` *string* [*start 0*] [*len* (*string-length string*)] [bigloo procedure]
Compute a hash value for *string*, starting at index *start*, ending at length *len*.

5.7 System programming

5.7.1 Operating System interface

`bigloo-config` [bigloo procedure]
`bigloo-config` *key* [bigloo procedure]
The function `bigloo-config` returns an alist representing the configuration of the running Bigloo system. When used with one parameter, the function `bigloo-config` returns the value associated with the key.

Examples:

```
(bigloo-config) ⇒ ((release-number . 3.4b) ... (endianess . little-endian))
(bigloo-config 'endianess) ⇒ little-endian
(bigloo-config 'int-size) ⇒ 61
```

`register-exit-function!` *proc* [bigloo procedure]
`unregister-exit-function!` *proc* [bigloo procedure]
Register *proc* as an exit functions. *Proc* is a procedure accepting of one argument. This argument is the numerical value which is the status of the exit call. The registered functions are called when the execution ends.

`exit` *int* [bigloo procedure]
Apply all the registered exit functions then stops an execution, returning the integer *int*.

`signal` *n proc* [bigloo procedure]
Provides a signal handler for the operating system dependent signal *n*. *proc* is a procedure of one argument.

`get-signal-handler` *n* [bigloo procedure]
Returns the current handler associated with signal *n* or `#f` if no handler is installed.

`system` . *strings* [bigloo procedure]
Append all the arguments *strings* and invoke the native host `system` command on that new string which returns an integer.

`system->string` . *strings* [bigloo procedure]
Append all the arguments *strings* and invoke the native host `system` command on that new string. If the command completes, `system->string` returns a string made of the output of the command.

`getenv` [*name*] [bigloo procedure]
Returns the string value of the Unix shell's *name* variable. If no such variable is bound, `getenv` returns `#f`. If *name* is not provided, `getenv` returns an alist composed of all the environment variables.

`putenv` *string val* [bigloo procedure]
Adds or modifies the global environment variable *string* so that it is bound to *val* after the call. This facility is not supported by all back-end. In particular, the JVM back-end does not support it.

`date` [bigloo procedure]
Returns the current date in a `string`. See also Section 5.8 [Date], page 95.

`sleep` *micros* [bigloo procedure]
Sleeps for a delay during at least *micros* microseconds.

`command-line` [bigloo procedure]
Returns a list of strings which are the Unix command line arguments.

`executable-name` [bigloo procedure]
Returns the name of the running executable.

`os-class` [bigloo procedure]
Gives the OS class (e.g. '`unix`').

`os-name` [bigloo procedure]
Gives the OS name (e.g. '`Linux`').

`os-arch` [bigloo procedure]
Gives the host architecture (e.g. '`i386`').

`os-version` [bigloo procedure]
Gives the operating system version (e.g. '`RedHat 2.0.27`').

`os-tmp` [bigloo procedure]
Gives the regular temporary directory (e.g. '`/tmp`').

`os-charset` [bigloo procedure]
Gives the charset used for encoding names of the file system (e.g. '`UTF-8`').

`file-separator` [bigloo procedure]
Gives the operating system file separator (e.g. '`#\/`').

`path-separator` [bigloo procedure]
Gives the operating system file path separator (e.g.'`#\:`').

For additional functions (such as `directory->list`) see Section 5.2 [Input and Output], page 54.

`unix-path->list` [bigloo procedure]
Converts a Unix path to a Bigloo list of strings.

```
(unix-path->list ".")           ⇒ (".")
(unix-path->list ".:/usr/bin")  ⇒ ("." "/usr/bin")
```

`hostname` [bigloo procedure]
Returns the fully qualified name of the current host.

`time` *thunk* [bigloo procedure]
Evaluates the *thunk* and returns four values: the result of calling *thunk*, the actual execution time, the system time, and the user time in millisecond.

```
(multiple-value-bind (res rtime stime utime)
  (time (lambda () (fib 35)))
  (print "real: " rtime " sys: " stime " user: " utime))
```

`getuid` [bigloo procedure]
`getgid` [bigloo procedure]
`setuid` *uid* [bigloo procedure]
`setgid` *uid* [bigloo procedure]
The procedure `getuid` (resp. `getgid`) returns the UID (resp. GID) of the user the current process is executed on behalf of.

The procedure `setuid` (resp. `setgid`) set the UID (resp. GID) of the current process. In case of failure, this procedure raises an error.

`getpid` [bigloo procedure]
Get the current process identifier.

`getppid` [bigloo procedure]
Get the parent process identifier.

`getgroups` [bigloo procedure]
Maps the Posix `getgroups` function, which returns the supplementary group IDs of the calling process. The result is a vector of IDs. On error, an IO exception is raised.

`getpwnam` *name* [bigloo procedure]
`getpwuid` *uid* [bigloo procedure]
These two procedures returns information about a user. The procedure `getpwname` accepts a string denoting the user name as argument. The procedure `getpwuid` accepts an UID as returned by the procedure `getuid`.

If the user is found, these two procedures returns a list of seven elements:

- the user name,
- his encrypted password,
- his uid,
- his group id,
- his real name,
- his home directory,
- his preferred shell.

When no user is found, these procedures returns `#f`.

`openlog` *name option facility* [bigloo procedure]
`syslog` *level . obj* [bigloo procedure]
`closelog` [bigloo procedure]
`syslog-option` [bigloo procedure]
`syslog-level` [bigloo procedure]
`syslog-facility` [bigloo procedure]
Wrapper to Unix syslog facilities. See the `syslog` man page for detail. Example.

```
(openlog "foo.scm" (syslog-option 'LOG_PID 'LOG_ODELAY) (syslog-facility 'LOG_MAIL))
(syslog (syslog-level 'LOG_INFO) "this is a log message")
(closelog)
```

5.7.2 Files

See Section 5.2 [Input and Output], page 54, for file and directory handling. This section only deals with *name* handling. Four procedures exist to manipulate Unix filenames.

`basename` *string* [bigloo procedure]
Returns a copy of *string* where the longest prefix ending in '`/`' is deleted if any existed.

`prefix` *string* [bigloo procedure]
Returns a copy of *string* where the suffix starting by the char '`#\.`' is deleted. If no prefix is found, the result of `prefix` is a copy of `string`. For instance:

```
(prefix "foo.scm")
   ⇒ "foo"
(prefix "./foo.scm")
```

```
   ⇒ "./foo"
(prefix "foo.tar.gz")
   ⇒ "foo.tar"
```

`suffix` *string* [bigloo procedure]

Returns a new string which is the suffix of *string*. If no suffix is found, this function returns an empty string. For instance,

```
(suffix "foo.scm")
   ⇒ "scm"
(suffix "./foo.scm")
   ⇒ "scm"
(suffix "foo.tar.gz")
   ⇒ "gz"
```

`dirname` *string* [bigloo procedure]

Returns a new string which is the directory component of *string*. For instance:

```
(dirname "abc/def/ghi")
   ⇒ "abc/def"
(dirname "abc")
   ⇒  "."
(dirname "abc/")
   ⇒ "abc"
(dirname "/abc")
   ⇒ "/"
```

`pwd` [bigloo procedure]

Returns the current working directory.

`chdir` *dir-name* [bigloo procedure]

Changes the current directory to *dir-name*. On success, `chdir` returns `#t`. On failure it returns `#f`.

`make-file-name` *dir-name name* [bigloo procedure]

Make an absolute file-name from a directory name *dir-name* and a relative name *name*.

`make-file-path` *dir-name name . names* [bigloo procedure]

Make an absolute file-name from a directory name *dir-name* and a relative name *name*s.

`file-name->list` *name* [bigloo procedure]

Explodes a file name into a list.

```
(file-name->list "/etc/passwd")
   ⇒ '("" "etc" "passwd")
(file-name->list "etc/passwd")
   ⇒ '("etc" "passwd")
```

`file-name-canonicalize` *name* [bigloo procedure]
`file-name-canonicalize!` *name* [bigloo procedure]
`file-name-unix-canonicalize` *name* [bigloo procedure]
`file-name-unix-canonicalize!` *name* [bigloo procedure]

Canonicalizes a file name. If the file name is malformed this function raises an `&io-malformed-url-error` exception.

The function `file-name-canonicalize!` may returns its argument if no changes in the string is needed. Otherwise, as `file-name-canonicalize` is returns a new string. In addition to handling `..` directory name, the function `file-name-unix-canonicalize` also handles the `~` character.

```
(file-name-canonicalize "/etc/passwd")
   ⇒ "/etc/passwd"
(file-name-canonicalize "/etc/../tmp/passwd")
   ⇒ "/tmp/passwd"
(file-name-canonicalize "~/passwd")
   ⇒ "~/passwd"
(file-name-unix-canonicalize "~/passwd")
   ⇒ "/home/a-user/passwd"
(file-name-unix-canonicalize "~foo/passwd")
   ⇒ "/home/foo/passwd"
```

`relative-file-name` *name base* [bigloo procedure]
Builds a file name relative to *base*.

```
(relative-file-name "/etc/passwd" "/etc"
   ⇒ "passwd"
```

`find-file/path` *name path* [bigloo procedure]
Search, in sequence, in the directory list *path* for the file *name*. If *name* is an absolute name, then *path* is not used to find the file. If *name* is a relative name, the function `make-file-name` is used to build absolute name from *name* and the directories in *path*. The current path is not included automatically in the list of *path*. In consequence, to check the current directory one may add `"."` to the *path* list. On success, the absolute file name is returned. On failure, `#f` is returned. Example:

```
(find-file/path "/etc/passwd" '("/toto" "/titi"))
   ⇒ "/etc/passwd"
(find-file/path "passwd" '("/toto" "/etc"))
   ⇒ "/etc/passwd"
(find-file/path "pass-wd" '("." "/etc"))
   ⇒ #f
```

`make-static-library-name` *name* [bigloo procedure]
Make a static library name from *name* by adding the static library regular suffix.

`make-shared-library-name` *name* [bigloo procedure]
Make a shared library name from *name* by adding the shared library regular suffix.

`file-exists?` *string* [bigloo procedure]
This procedure returns `#t` if the file (respectively directory, and link) *string* exists. Otherwise it returns `#f`.

`file-gzip?` *string* [bigloo procedure]
This procedure returns `#t` if and only if the file *string* exists and can be unzip by Bigloo. Otherwise it returns `#f`.

`delete-file` *string* [bigloo procedure]
Deletes the file named *string*. The result of this procedure is `#t` is the operation succeeded. The result is `#f` otherwise.

`rename-file` *string1 string2* [bigloo procedure]
Renames the file *string1* as *string2*. The two files have to be located on the same file system. If the renaming succeeds, the result is `#t`, otherwise it is `#f`.

`truncate-file` *path size* [bigloo procedure]
Truncates shall cause the regular file named by path to have a size which shall be equal to length bytes.

Returns `#t` on success. Returns `#f` otherwise.

`copy-file` *string1 string2* [bigloo procedure]
Copies the file *string1* into *string2*. If the copy succeeds, the result is `#t`, otherwise it is `#f`.

`make-symlink` *target linkpath* [bigloo procedure]
Creates a symbolic link named *linkpath* which contains the string *target*.

`directory?` *string* [bigloo procedure]
This procedure returns `#t` if the file *string* exists and is a directory. Otherwise it returns `#f`.

`make-directory` *string* [bigloo procedure]
Creates a new directory named *string*. It returns `#t` if the directory was created. It returns `#f` otherwise.

`make-directories` *string* [bigloo procedure]
Creates a new directory named *string*, including any necessary but nonexistent parent directories. It returns `#t` if the directory was created. It returns `#f` otherwise. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

`delete-directory` *string* [bigloo procedure]
Deletes the directory named *string*. The directory must be empty in order to be deleted. The result of this procedure is unspecified.

`directory->list` *string* [bigloo procedure]
`directory->path-list` *string* [bigloo procedure]
If file *string* exists and is a directory, the function `directory->list` returns the list of files in *string*. The function `directory->path-list` returns a list of files whose dirname is *string*.

`file-modification-time` *string* [bigloo procedure]
`file-change-time` *string* [bigloo procedure]
`file-access-time` *string* [bigloo procedure]
`file-times-set!` *string atime mtime* [bigloo procedure]
The date (in second) of the last modification (respec. access) for file *string*. The number of seconds is represented by a value that may be converted into a date by the means of `seconds->date` (see Section 5.8 [Date], page 95).

`file-size` *string* [bigloo procedure]

Returns the size (in bytes) for file *string*. The return type is `long`. If an full-sized integer is needed, one may write:

```
(let ((sz::llong (file-size <PATH>)))
 ...)
```

`On error, -1 is returned.`

`file-uid` *string* [bigloo procedure]

`file-gid` *string* [bigloo procedure]

The functions return the user id (an integer) and group id (an integer) for file *string*. On error, `-1` is returned.

`file-mode` *string* [bigloo procedure]

Returns the file access mode (an integer). On error `-1` is returned.

`file-type` *string* [bigloo procedure]

Returns the file type (a symbol). The possible returned values are:

- `regular`
- `directory`
- `link`
- `block`
- `fifo`
- `character`
- `socket`
- `resource`
- `unknown`
- `does-not-exist`

`chmod` *string* [*option*] [bigloo procedure]

Change the access mode of the file named *string*. The *option* must be either a list of the following symbols `read`, `write` and `execute` or an integer. If the operation succeeds, `chmod` returns `#t`. It returns `#f` otherwise. The argument *option* can also be an integer that represents the native file permission. Example:

```
(chmod (make-file-name (getenv "HOME") ".bigloorc") 'read 'write)
(chmod (make-file-name (getenv "HOME") ".bigloorc") #o777)
```

5.7.3 Process support

Bigloo provides access to Unix-like processes as first class objects. The implementation and this documentation are to a great extent copies of the STk [Gallesio95] process support. Basically, a process contains four informations: the standard Unix process identification (aka PID) and the three standard files of the process.

`run-process` *command arg...* [bigloo procedure]

`run-process` creates a new process and run the executable specified in *command*. The *arg* correspond to the command line arguments. When is process completes its execution, non pipe associated ports are automatically closed. Pipe associated ports

have to be explicitly closed by the program. 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 ran asynchronously or not. By default, the process is run asynchronously (i.e. `wait:` if `#f`).
- `host:` must be followed by a string. This string represents the name of the machine on which the *command* must be executed. This option uses the external command `rsh`. The shell variable `PATH` must be correctly set for accessing it without specifying its absolute path.
- `fork:` must be followed by a boolean value. This value specifies if the process must substitute the current execution. That is, if the value is `#t` a new process is spawned otherwise, the current execution is stopped and replaced by the execution of `command`. It defaults to `#t`.
- `env:` must be followed by a string of the form *`var=val`*. This will bound an environment variable in the spawned process. A `run-process` command may contain several `env:` arguments. The current variables of the current process are also passed to the new process.

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")
```

The same example with a pipe for output:

```
(let* ((proc (run-process "ls" "-l" "/bin" output: pipe:))
       (port (process-output-port proc)))
   (let loop ((line (read-line port)))
      (if (eof-object? line)
          (close-input-port port)
          (begin
             (print line)
             (loop (read-line port))))))
```

One should note that the same program can be written with explicit process handling but making use of the `|` notation for `open-input-file`.

```
(let ((port (open-input-file "| ls -l /bin")))
   (let loop ((line (read-line port)))
```

```
      (if (eof-object? line)
          (close-input-port port)
          (begin
             (print line)
             (loop (read-line port))))))
```

Both input and output ports can be piped:

```
(let* ((proc (run-process "/usr/bin/dc" output: pipe: input: pipe:))
       (inport (process-input-port proc))
       (port (process-output-port proc)))
  (fprint inport "16 o")
  (fprint inport "16 i")
  (fprint inport "10")
  (fprint inport "10")
  (fprint inport "+ p")
  (flush-output-port inport)
  (let loop ((line (read-line port)))
     (if (eof-object? line)
(close-input-port port)
(begin
   (print line)
   (loop (read-line port))))))    ⊣ 20
```

Note: The call to `flush-output-port` is mandatory in order to get the `dc` process to get its input characters.

Note: Thanks to Todd Dukes for the example and the suggestion of including it this documentation.

`process?` *obj* [bigloo procedure]
Returns `#t` if *obj* is a process, otherwise returns `#f`.

`process-alive?` *process* [bigloo procedure]
Returns `#t` if *process* is currently running, otherwise returns `#f`.

`close-process-ports` *command arg...* [bigloo procedure]
Close the three ports associated with a process. In general the ports should not be closed before the process is terminated.

`process-pid` *process* [bigloo procedure]
Returns an integer value which represents the Unix identification (PID) of the *process*.

`process-input-port` *process* [bigloo procedure]
`process-output-port` *process* [bigloo procedure]
`process-error-port` *process* [bigloo procedure]
Return the file port associated to the standard input, output and error of *process* otherwise returns `#f`. Note that the returned port is opened for reading when calling `process-output-port` or `process-error-port`. It is opened for writing when calling `process-input-port`.

`process-wait` *process* [bigloo procedure]
This function stops the current process until *process* completion. This function returns `#f` when *process* is already terminated. It returns `#t` otherwise.

`process-exit-status` *process* [bigloo procedure]
This function returns the exit status of *process* if it is has finished its execution. It returns `#f` otherwise.

`process-send-signal` *process s* [bigloo procedure]
Sends the signal whose integer value is *s* to *process*. Value of *s* is system dependent. The result of `process-send-signal` is undefined.

`process-kill` *process* [bigloo procedure]
This function brutally kills *process*. The result of `process-kill` is undefined.

`process-stop` *process* [bigloo procedure]
`process-continue` *process* [bigloo procedure]
Those procedures are only available on systems that support job control. The function `process-stop` stops the execution of *process* and `process-continue` resumes its execution.

`process-list` [bigloo procedure]
This function returns the list of processes which are currently running (i.e. alive).

5.7.4 Socket support

Bigloo defines sockets, on systems that 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. The implementation and this documentation are, to a great extent copies of the STk [Gallesio95] socket support.

Bigloo supports both *stream-oriented* sockets and *datagram* sockets (see Section "The GNU C Library Reference Manual" in `libc`). Stream-oriented sockets are created and manipulated with the following procedures.

`make-client-socket` *hostname port-number #!key* (*timeout 0*) (*inbuf #t*) (*outbuf #t*) (*domain 'inet*) [bigloo 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*. If keyword arguments *inbuf* and *outbuf* describe the buffer to be used. Each can either be:

- A positive fixnum, this gives the size of the buffer.
- The boolean `#t`, a buffer is allocated by the Bigloo runtime system with a default size.
- The boolean `#f`, the socket is unbufferized.
- A string, it is used as buffer.

Unbuffered sockets are useful for socket clients connected to servers that do not emit #\Newline character after emissions. If the optional argument *timeout* is missing or is `0`, the execution blocks until the connection is established. If the *timeout* is provided, the execution unblocks after *timeout* microseconds unless the connection is established.

The *domain* argument specifies the protocol used by the socket. The supported domains are:

- `inet`: IPv4 Internet protocols.

- `unix`: Unix sockets for local inter-process communications.
- `local`: Same as `unix`.

If the connection cannot be established, an `&io-error` is raised (see Chapter 15 [Errors Assertions and Traces], page 173).

When a socket is used in unbufferized mode the characters available on the input port *must* be read exclusively with `read-char` or `read-line`. It is forbidden to use `read` or any regular grammar. This limitation is imposed by Rgc (see Chapter 10 [Regular Parsing], page 129) that intrinsicly associates buffers with regular grammars. If the current Rgc implementation is improved on the coming version this restriction will be eliminated.

Example:

```
;; open a client socket on port 80:
(make-client-socket "www.inria.fr" 80)
;; open an unbufferized connection
(make-client-socket "www.inria.fr" 80 :inbuf #f :outbuf #f)
```

`socket?` *obj* [bigloo procedure]
`socket-server?` *obj* [bigloo procedure]
`socket-client?` *obj* [bigloo procedure]
Returns `#t` if *obj* is a socket, a socket server a socket client. Otherwise returns `#f`. Socket servers and socket clients are sockets.

`socket-hostname` *socket* [bigloo 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* [bigloo 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* [bigloo procedure]
Returns a string which contains the IP number of the local host attached to *socket*.

`socket-port-number` *socket* [bigloo procedure]
Returns the integer number of the port used for *socket*.

`socket-input` *socket* [bigloo procedure]
`socket-output` *socket* [bigloo 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`"[1]:

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (print "Time is: " (read-line (socket-input s)))
  (socket-shutdown  s))
```

`make-server-socket` *#!optional* (*port 0*) *#!key* (*name #f*) (*backlog 5*) [bigloo procedure]

`make-server-socket` returns a new socket object. The socket will be listening on the network interface *name*, either on the specified *port*, or on a port chosen by the system (usually the first port available on the network interface). The *name* can be an IP number as a string, or a host name, whose first IP address will be used (as returned by the name server lookup).

The *backlog* argument specifies the size of the wait-queue used for accepting connections.

`socket-accept` *socket #!key* (*errp #t*) (*inbuf #t*) (*outbuf #t*) [bigloo procedure]

`socket-accept` waits for a client connection on the given *socket*. It returns a `client-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 arguments *inbuf* and *outbuf* are similar to the ones used by `make-client-socket`. That is, each can either be:

- A positive fixnum, this gives the size of the buffer.
- The boolean `#t`, a buffer is allocated.
- The boolean `#f`, the socket is unbufferized.
- A string, it is used as buffer.

The keyword argument *errp* is a boolean. The value `#t` means that if an error is raised it is signaled. Otherwise, it is omitted.

Note: When a socket is used in unbufferized mode the characters available on the input port *must* be read exclusively with `read-char` or `read-line`. It is forbidden to use `read` or any regular grammar. This limitation is imposed by Rgc (see Chapter 10 [Regular Parsing], page 129) that intrinsicly associate buffers with regular grammars. If the current Rgc implementation is improved on the coming version this restriction will be suppressed.

The following exemple is a simple server which waits for a connection on the port 1234[2]. 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))
```

[1] 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.

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

```
        (s2 (socket-accept s)))
    (let ((l (read-line (socket-input s2))))
      (fprint (socket-output s2) "Length is: " (string-length l))
      (flush-output-port (socket-output s2)))
    (socket-close s2)
    (socket-shutdown s))
```

`socket-close` *socket* [bigloo procedure]

The function `socket-close` closes the connection established with a `socket-client`.

`socket-shutdown` *socket #!optional* (*how #t*) [bigloo procedure]

`Socket-shutdown` shutdowns the connection associated to *socket*.

Close is either a boolean or one of the symbols `RDWR`, `RD`, or `WR`. The meaning of the optional how (which defaults to `#t`) is as follows:

- `#t`, the socket is shutdown for reading and writing *and* the socket is closed.
- `#f`, the socket is shutdown for reading and writing.
- `RDWR`, the socket is shutdown for reading and writing.
- `RD`, the socket is shutdown for reading.
- `WD`, the socket is shutdown for writing.

The function `socket-shutdown` returns an integer which is `0` is the operation has succeeded and a positive integer otherwise.

`socket-down?` *socket* [bigloo procedure]

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

Here is another example of making use of stream sockets:

```
(define s1 (make-server-socket))
(define s2 #unspecified)

(dynamic-wind
   ;; Init: Launch an xterm with telnet running
   ;; on the s listening port and connect
   (lambda ()
      (run-process "/usr/X11R6/bin/xterm" "-display" ":0" "-e" "telnet" "localhost"
   (number->string (socket-port-number s1)))
      (set! s2 (socket-accept s1))
      (display #"\nWelcome on the socket REPL.\n\n> " (socket-output s2))
      (flush-output-port (socket-output s2)))

   ;; Action: A toplevel like loop
   (lambda ()
      (let loop ()
 (let ((obj (eval (read (socket-input s2)))))
    (fprint (socket-output s2) "; Result: " obj)
    (display "> " (socket-output s2))
    (flush-output-port (socket-output s2))
    (loop))))

   ;; Termination: We go here when
   ;;     -a: an error occurs
   ;;     -b: connection is closed
```

```
   (lambda ()
      (print #"Shutdown ......\n")
      (socket-close s2)
      (socket-shutdown s1)))
```

Here is a second example that uses sockets. It implements a client-server architecture and it uses unbufferized (see `socket-accept`) input ports.

First, here is the code of the client:

```
(module client)

(let* ((s (make-client-socket "localhost" 8080 :outbuf #f))
       (p (socket-output s)))
   (display "string" p)
   (newline p)
   (display "abc" p)
   (flush-output-port p)
   (let loop ()
      (loop)))
```

Then, here is the code of the server:

```
(module server)

(let* ((s (make-server-socket 8080))
       (s2 (socket-accept s :inbuf #f)))
   (let ((pin (socket-input s2)))
      (let loop ()
         (display (read-char pin))
         (flush-output-port (current-output-port))
         (loop))))
```

At, to conclude here the source code for a server waiting for multiple consecutive connections:

```
(define (main argv)
   (let ((n (if (pair? (cdr argv))
                (string->integer (cadr argv))
                10))
 (s (make-server-socket)))
      (print "s: " s)
      (let loop ((i 0))
         (if (<fx i n)
             (let ((s2 (socket-accept s)))
(print "i: " i " " s2)
(print (read-line (socket-input s2)))
(socket-close s2)
                (loop (+fx i 1)))
     (socket-shutdown s)))))
```

Bigloo also provides primitives dealing with *datagram* sockets, for use with transports such as UDP. These are shown below:

`make-datagram-server-socket` *port* [bigloo procedure]

Return a datagram server socket bound to the loopback address on *port*, and whose address family and protocol family are those normally used for services on *port*.

`make-datagram-unbound-socket` [(*domain 'inet*)] [bigloo procedure]

Return an unbound datagram socket. It may then be used in conjunction with `datagram-socket-send` and `datagram-socket-receive`, for instance send to and receive from a UDP multicast address.

`datagram-socket-receive` *sock size* [bigloo procedure]
Receive up to *size* bytes from datagram socket *sock*, and return them as a string.

`datagram-socket-send` *sock message host port* [bigloo procedure]
Send string *message* over datagram socket *sock* to *host* and *port*. *host* must be a string denoting an IPv4 or IPv6 address. On success, return the number of bytes actually sent.

`host` *hostname* [bigloo procedure]
`hostinfo` *hostname* [bigloo procedure]
Returns the IP number of *hostname*. When *hostname* is not found, the `io-unknown-host-error` exception is raided (see Chapter 15 [Errors Assertions and Traces], page 173).

The function `hostinfo` possibly returns more information about the host. It returns an association list made out the information about the host. This list might contain a `name` entry, an `addresses` entry, and a `aliases` entry.

Some back-ends (e.g., the C back-end) implements DNS caching. This may dramatically improve the performance of intensive networking applications. DNS caching can be control by the means of two parameters: `bigloo-dns-enable-cache` and `bigloo-dns-cache-validity-timeout` (see Chapter 24 [Parameters], page 231).

`get-interfaces` [bigloo procedure]
Returns the list of configured interfaces, their associated IP addresses, their protocol, and, if supported by the system, the hardware address (the mac address).

`get-protocols` [bigloo procedure]
Reads all the entries from the protocols database and returns a list of protocol entries. Each entries consists in a list of three elements:

- a string denoting the protocol name,
- an integer denoting the protocol number,
- a list of strings denoting the protocol aliases.

`get-protocol` *number-or-name* [bigloo procedure]
Returns the protocol entry found in the protocols database. The argument *number-of-name* is either an integer or a string.

`socket-option` *socket option-name* [bigloo procedure]
`socket-option-set!` *socket option-name val* [bigloo procedure]
These two functions *get* and *set* socket option. The argument *option-name* must be a keyword. If the *option-name* is not supported by the Bigloo runtime system, the function `socket-option` returns the value `#unspecified` otherwise, it returns the option value. If the *option-name* is not supported, the function `socket-option-set!` returns `false`. Otherwise it returns a non false value.

Here is a list of possibly supported option-name values:

- `:SO_KEEPALIVE`
- `:SO_OOBINLINE`
- `:SO_RCVBUF`

- `:SO_SNDBUF`
- `:SO_REUSEADDR`
- `:SO_TIMEOUT`
- `:SO_SNDTIMEO`
- `:SO_RCVTIMEO`
- `:TCP_CORK`
- `:TCP_QUICKACK`
- `:TCP_NODELAY`

The `:SO_KEEPALIVE` option can be use to implement automatic notification of client disconnection. It requires system tuning for enabling TCP keeplive support. On Linux additional information may be found on the "TCP Keepalive HOWTO" (see `http://tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/`).

5.7.5 SSL

Bigloo allows access to SSL sockets, certificates and private keys, in order to build secure encrypted and/or signed communications.

`ssl-version` [SSL library procedure]
Returns a string representing the SSL library version number.

5.7.5.1 SSL Sockets

Bigloo defines SSL sockets, on systems that support them, as first class objects. SSL Sockets permits processes to communicate even if they are on different machines securely via encrypted connections. SSL Sockets are useful for creating secure client-server applications.

`ssl-socket?` *obj* [SSL library procedure]
Returns `#t` if an only if obj is a SSL socket (either client or server). Returns `#f` otherwise.

`make-ssl-client-socket` *hostname port-number #!key* [SSL library procedure]
(*buffer #t*) (*timeout 0*) (*protocol 'sslv23*) (*cert #f*) (*pkey #f*) (*CAs '()*) (*accepted-certs #f*)
`make-ssl-client-socket` returns a new client socket object. This object satisfies the `socket?` predicate (see Section 5.7.4 [Socket], page 86) can be used in any context where a socket created by `make-client-socket` can be used.

A SSL client socket establishes a link between the running application (client) and a remote application (server) listening on port *port-number* of *hostname*. If optional argument *bufsiz* is lesser or equal to `1` then the input port associated with the socket is unbuffered. This is useful for socket clients connected to servers that do not emit #\Newline character after emissions. The optional argument *buffer* can either be:

- A positive fixnum, this gives the size of the buffer.
- The boolean `#t`, a buffer is allocated.
- The boolean `#f`, the socket is unbufferized.
- A string, it is used as buffer.

If the optional argument *timeout* is `0`, the execution blocks until the connection is established. If the *timeout* is provided, the execution unblocks after *timeout* microseconds unless the connection is established. If the *protocol* option argument is given, it specifies the encryption protocol. Accepted values are `'sslv2`, `'sslv3`, `'sslv23` (alias `'ssl`), `'tlsv1` (alias `'tls`), `'tlsv1_1`, `'tlsv1_2`, `'tlsv1_3`, or `'dtlsv1` (alias `'dtls`). The default value is `'sslv23`.

The SSL socket will sign the connection using the optional arguments *cert* (for the certificate) and *pkey* (for the private key). The certificate *cert* must be of type `certificate`, and the private key *pkey* must be of type `private-key`. If any of those two arguments is given, they must both be given. If those optional arguments are missing the connection will be encrypted but not signed from the client side.

The *CAs* optional argument specifies the list of certificates to trust as CA (Certificate Authority) for the connection. It must be a list of values of type `certificate`. If the list is empty, the default list of trusted CA is used (set by the system). Note that giving a list of trusted certificates turns on the peer (server) certificate validation: an `&io-error` will be raised if the peer (server) certificate is not signed directly or indirectly by one of the certificates in *CAs*.

The *accepted-certs* optional argument gives a list of certificate objects (of type `certificate`) which are accepted as peer (server) certificate. If *accepted-certs* is `#f` then every peer (server) certificate is accepted (aside from eventual certificate validation). If *accepted-certs* is a list, the peer (server) certificate must match one of the given certificates. Otherwise, an `&io-error` will be raised.

If the connection cannot be established, an `&io-error` is raised (see Chapter 15 [Errors Assertions and Traces], page 173).

When a socket is used in unbufferized mode the characters available on the input port *must* be read exclusively with `read-char` or `read-line`. It is forbidden to use `read` or any regular grammar. This limitation is imposed by Rgc (see Chapter 10 [Regular Parsing], page 129) that intrinsicly associates buffers with regular grammars. If the current Rgc implementation is improved on the coming version this restriction will be eliminated.

The function `make-ssl-client-socket` is defined in the SSL library. A module that needs this facility must then use a `library` clause (see Chapter 2 [Modules], page 7). The SSL library can also be loaded from the interpreter using the `library-load` function (see Chapter 28 [Bigloo Libraries], page 255).

```
(module imap
   (library ssl)
   (main main))

(let* ((s (make-ssl-client-socket "localhost" 993))
       (p (socket-output s)))
   (display "string" p)
   (newline p)
   (display "abc" p)
   (flush-output-port p)
   (let loop ()
      (loop)))
```

`client-socket-use-ssl!` *socket #!key* (*protocol 'sslv23*) (*cert #f*) (*pkey #f*) (*CAs '()*) (*accepted-certs #f*) [SSL library procedure]

Returns an SSL socket built from a socket obtained by `make-client-socket` (see Section 5.7.4 [Socket], page 86). Depending on the implementation and back-end the returned socket may or may not be `eq?` to *socket*.

`make-ssl-server-socket` *#!key* (*port 0*) (*name #f*) (*protocol 'sslv23*) (*cert #f*) (*pkey #f*) (*CAs '()*) (*accepted-certs #f*) [SSL library procedure]

`make-ssl-server-socket` returns a new server socket object which satisfies the `socket?` predicate and which can be used in any context where a socket created by `make-server-socket` can be used (see Section 5.7.4 [Socket], page 86).

A SSL server socket opens the port *port* on the current host *name* (the server), and allows remote applications (clients) to connect to it. listening on port *port-number* of *hostname*. If the optional argument *port* is not given or is `0`, the server socket will use the first availailable port number. If the optional argument *name* is given, the server socket will be bound to the network interface representing the given host name. If it is `#f` (the default) the socket will be bound on every local network interface. If the *protocol* option argument is given, it specifies the encryption protocol. Accepted values are `'sslv2`, `'sslv3`, `'sslv23` (alias `'ssl`), `'tlsv1` (alias `'tls`), `'tlsv1_1`, `'tlsv1_2` `'tlsv1_3`, or `'dtlsv1` (alias `'dtls`). The default value is `'sslv23`.

The SSL socket will sign the connection using the optional arguments *cert* (for the certificate) and *pkey* (for the private key). The certificate *cert* must be of type `certificate`, and the private key *pkey* must be of type `private-key`. If any of those two arguments is given, they must both be given. If those optional arguments are missing the connection will be encrypted but not signed from the server side, which means the peer (client) will have to provide a certificate/private key pair to encrypt the connection, and that seldom happens. Typical SSL servers provide their certificate and private key.

Note that since the peer (client) certificate is only known when we are accepting a client socket (with `socket-accept`) the *CAs* and *accepted-certs* optional arguments are only checked during the accept operation of a server socket.

The *CAs* optional argument specifies the list of certificates to trust as CA (Certificate Authority) for the connection. It must be a list of values of type `certificate`. If the list is empty, the default list of trusted CA is used (set by the system). Note that giving a list of trusted certificates turns on the peer (client) certificate validation: an `&io-error` will be raised if the peer (client) certificate is not signed directly or indirectly by one of the certificates in *CAs* when accepting the client socket.

The *accepted-certs* optional argument gives a list of certificate objects (of type `certificate`) which are accepted as peer (client) certificate. If *accepted-certs* is `#f` then every peer (client) certificate is accepted (aside from eventual certificate validation). If *accepted-certs* is a list, the peer (client) certificate must match one of the given certificates. Otherwise, an `&io-error` will be raised when accepting the client socket.

If the connection cannot be established, an `&io-error` is raised (see Chapter 15 [Errors Assertions and Traces], page 173).

The function `make-ssl-server-socket` is defined in the SSL library. A module that needs this facility must then use a `library` clause (see Chapter 2 [Modules], page 7). The SSL library can also be loaded from the interpreter using the `library-load` function (see Chapter 28 [Bigloo Libraries], page 255).

```
(module secure-echo
   (library ssl))

(let* ((cert (read-certificate "/etc/ssl/my_cert.crt"))
       (pkey (read-private-key "/etc/ssl/my_key.pkey"))
       (cas (read-pem-file "/etc/ssl/ca.cert"))
       (s (make-ssl-server-socket 1055 :CAs cas :cert cert :pkey pkey))
       (cs (socket-accept s))
       (ip (socket-input cs))
       (op (socket-output cs)))
   (let loop ((e (read ip)))
      (when (not (eof-object? e))
         (write e op)
         (loop (read ip))))
   (socket-close s))
```

5.7.5.2 Certificates

Certificates are instances of the `certificate` class. There type can be checked with `(isa? expr certificate)`.

`read-certificate` *file* [SSL library procedure]
Reads an X509 certificate stored in PEM format in the given *file* name. If the file cannot be read, it raises an `&io-error` condition. Otherwise the certificate is returned.

`read-pem-file` *file* [SSL library procedure]
Reads a list of X509 certificate stored in PEM format in the given *file* name. If the file cannot be read, it raises an `&io-error` condition. Otherwise the list of certificate contained in the file is returned.

`certificate-subject` *cert* [SSL library procedure]
Returns the CommonName (CN) part of the subject of the given certificate.

`certificate-issuer` *cert* [SSL library procedure]
Returns the CommonName (CN) part of the issuer of the given certificate.

5.7.5.3 Private Keys

Private keys are instances of the `private-key` class. There type can be checked with `(isa? expr private-key)`.

`read-private-key` *file* [SSL library procedure]
Reads a private key stored in PEM format in the given *file* name. If the file cannot be read, it raises an `&io-error` condition. Otherwise the private key is returned.

5.8 Date

`date?` *obj* [bigloo procedure]
Returns `#t` if and only if *obj* is a *date* as returned by `make-date`, `current-date`, or `seconds->date`. It returns `#f` otherwise.

`make-date` *#!key* (*nsec 0*) (*sec 0*) (*min 0*) (*hour 0*) (*day 1*) (*month 1*) (*year 1970*) *timezone* (*dst -1*) [bigloo procedure]

Creates a `date` object from the integer values passed as argument.

The argument *timezone*, if provided, is expressed in minute.

Example:

```
(write (make-date :sec 0 :min 22 :hour 17 :day 5 :month 2 :year 2003 :dst 0))
  ⊣ #<date:Wed Feb  5 17:22:00 2003>
```

The argument *dst* is either `-1` when the information is not available, `0` when daylight saving is disabled, `1` when daylight saving is enabled.

`date-copy` *date #!key sec min hour day month year timezone* [bigloo procedure]

Creates a new date from the argument *date*.

Example:

```
(date-copy (current-date) :sec 32 :min 24 :day 5)
```

`current-date` [bigloo procedure]

Returns a `date` object representing the current date.

`current-seconds` [bigloo procedure]
`current-microseconds` [bigloo procedure]
`current-nanoseconds` [bigloo procedure]

Returns an `elong` integer representing the current epoch (i.e., the date since 0:00:00 UTC on the morning of 1 January 1970, expressed in seconds (resp. in micro seconds).

`date->seconds` [bigloo procedure]
`date->nanoseconds` [bigloo procedure]
`seconds->date` [bigloo procedure]
`nanoeconds->date` [bigloo procedure]

Convert from `date` and `elong`.

`date->string` *date* [bigloo procedure]
`date->utc-string` *date* [bigloo procedure]
`seconds->string` *elong* [bigloo procedure]
`seconds->utc-string` *elong* [bigloo procedure]

Construct a textual representation of the date passed in argument

`date-second` *date* [bigloo procedure]

Returns the number of seconds of a date, in the range `0...59`.

`date-nanosecond` *date* [bigloo procedure]

Returns the number of nano seconds of a date (to be added to `date-second`).

`date-minute` *date* [bigloo procedure]

Returns the minute of a date, in the range `0...59`.

`date-hour` *date* [bigloo procedure]

Returns the hour of a date, in the range `0...23`.

`date-day` *date* [bigloo procedure]

Returns the day of a date, in the range `1...31`.

`date-wday` *date* [bigloo procedure]
`date-week-day` *date* [bigloo procedure]
Returns the week day of a date, in the range `1...7`.

`date-yday` *date* [bigloo procedure]
`date-year-day` *date* [bigloo procedure]
Returns the year day of a date, in the range `1...366`.

`date-month` *date* [bigloo procedure]
Returns the month of a date, in the range `1...12`.

`date-year` *date* [bigloo procedure]
Returns the year of a date.

`date-timezone` *date* [bigloo procedure]
Returns the timezone (in seconds) of a date.

`date-is-dst` *date* [bigloo procedure]
Returns `-1` if the information is not available, `0` is the date does not contain daylight saving adjustment, `1` if it contains a daylight saving adjustment.

`integer->second` [bigloo procedure]
Converts a Bigloo fixnum integer into a second number.

`day-seconds` [bigloo procedure]
Returns the number of seconds contained in one day.

`day-name` *int* [bigloo procedure]
`day-aname` *int* [bigloo procedure]
Return the name and the abbreviated name of a week day.

`month-name` *int* [bigloo procedure]
`month-aname` *int* [bigloo procedure]
Return the name and the abbreviated name of a month.

`date-month-length` *date* [bigloo procedure]
Return the length of the month of *date*.

`leap-year?` *int* [bigloo procedure]
Returns `#t` if and only if the year *int* is a leap year. Returns `#f` otherwise.

`rfc2822-date->date` *string* [bigloo procedure]
`rfc2822-parse-date` *input-port* [bigloo procedure]
Parses RFC2822 string representing a date. These functions produce a Bigloo date object.

`date->rfc2822-date` *date* [bigloo procedure]
Converts a Bigloo date into a string representation compliant with the RFC2822 format.

`iso8601-date->date` *string* [bigloo procedure]
`iso8601-parse-date` *input-port* [bigloo procedure]

Parses ISO8601 string representing a date. These functions produce a Bigloo date object.

`date->iso8601-date` *date* [bigloo procedure]

Converts a Bigloo date into a string representation compliant with the iso8601 format.

5.9 Digest

`base64-encode` *string* [*padding 64*] [bigloo procedure]
`base64-decode` *string* [*no-eof-padding*] [bigloo procedure]

Encodes (respec. decodes) a string into a base64 representation.

When decoding, if the optional parameter *no-eof-padding* is `#t`, the decoding success even if the input stream is not padded with `=` characters.

`base64-encode-port` *input-port output-port* [*padding 64*] [bigloo procedure]
`base64-decode-port` *input-port output-port* [*no-eof-padding*] [bigloo procedure]

Encodes (respec. decodes) an input port into a base64 representation.

When decode succeeds, *base64-decode-port* returns `#t`, it returns `#f` otherwise.

When decoding, if the optional parameter *no-eof-padding* is `#t`, the decoding success even if the input stream is not padded with `=` characters.

`pem-read-file` *file-name* [bigloo procedure]
`pem-decode-port` *input-port output-port* [bigloo procedure]

Reads a PEM (Privacy Enhanced Mail) base64 encoded file.

`md5sum` *obj* [bigloo procedure]
`md5sum-string` *string* [bigloo procedure]
`md5sum-mmap` *mmap* [bigloo procedure]
`md5sum-file` *string* [bigloo procedure]
`md5sum-port` *input-port* [bigloo procedure]

Computes MD5 message digest.

The function `md5sum` dispatches over its argument and invokes the ad-hoc function. That is, it invokes `md5sum-string` if its argument is a string, `md5sum-mmap` if it is a mmap, `md5sum-port` if its argument is an input port.

`hmac-md5sum-string` *key string* [bigloo procedure]

Computes the Hmac MD5 authentication:

```
(hmac-md5sum-string (make-string 16 #a011) "Hi There")
  ⇒ "9294727a3638bb1c13f48ef8158bfc9d"
```

`cram-md5sum-string` *user key string* [bigloo procedure]

Challenge-Response Authentication Mechanism as specified in RFC 2195.

The function `cram-md5sum-string` assumes that data is base64 encoded. The result is also base64 encoded.

`sha1sum` *obj* [bigloo procedure]
`sha1sum-string` *string* [bigloo procedure]
`sha1sum-mmap` *mmap* [bigloo procedure]
`sha1sum-file` *string* [bigloo procedure]
`sha1sum-port` *input-port* [bigloo procedure]

Computes SHA1 message digest.

The function `sha1sum` dispatches over its argument and invokes the ad-hoc function. That is, it invokes `sha1sum-string` if its argument is a string, `sha1sum-mmap` if it is a mmap, `sha1sum-port` if its argument is an input port.

`hmac-sha1sum-string` *key string* [bigloo procedure]

Computes the Hmac SHA1 authentication:

`sha256sum` *obj* [bigloo procedure]
`sha256sum-string` *string* [bigloo procedure]
`sha256sum-mmap` *mmap* [bigloo procedure]
`sha256sum-file` *string* [bigloo procedure]
`sha256sum-port` *input-port* [bigloo procedure]

Computes SHA256 message digest.

The function `sha256sum` dispatches over its argument and invokes the ad-hoc function. That is, it invokes `sha256sum-string` if its argument is a string, `sha256sum-mmap` if it is a mmap, `sha256sum-port` if its argument is an input port.

`hmac-sha256sum-string` *key string* [bigloo procedure]

Computes the Hmac SHA256 authentication:

5.10 Cyclic Redundancy Check (CRC)

Bigloo provides several known cyclic redundancy checks as well as means to create custom checks.

Usually CRCs are executed starting with the leftmost bit inside a byte (big endian). However, especially for serial-port transmissions, a scheme where the least-significant bit is processed first is desirable. Bigloo's CRC procedures accept a key-parameter (`:big-endian`) (by default `#t`) which allows to change this behavior.

The following CRCs (given with the associated polynomial) are provided:

- `itu-4`: 0x3
- `epc-5`: 0x9
- `itu-5`: 0x15
- `usb-5`: 0x5
- `itu-6`: 0x3
- `7`: 0x9
- `atm-8`: 0x7
- `ccitt-8`: 0x8d
- `dallas/maxim-8`: 0x31
- `8`: 0xd5

- `sae-j1850-8`: 0x1d
- `10`: 0x233
- `11`: 0x385
- `12`: 0x80f
- `can-15`: 0x4599
- `ccitt-16`: 0x1021
- `dnp-16`: 0x3d65
- `ibm-16`: 0x8005
- `24`: 0x5d6dcb
- `radix-64-24`: 0x864cfb
- `30`: 0x2030b9cf
- `ieee-32`: 0x4c11db7
- `c-32`: 0x1edc6f41
- `k-32`: 0x741b8cd7
- `q-32`: 0x814141ab
- `iso-64`: 0x1b
- `ecma-182-64`: 0x42f0e1eba9ea3693

`crc-names` [bigloo procedure]
Returns a list of all provided CRCs (`itu-4`, `epc-5`, etc.).

`crc-polynomial` *name* [bigloo procedure]
`crc-polynomial-le` *name* [bigloo procedure]
Returns the polynomial for the given name. The `-le` variant returns the little endian polynomial.

```
(crc-polynomial 'ieee-32)
    ⊣ #e79764439 ;; == #ex4c11bd7
(crc-polynomial 24)
    ⊣ 6122955    ;; == #x5d6dcb
```

`crc-length` *name* [bigloo procedure]
Returns the length of the specified CRC.

`crc` *name obj* [*:init 0*] [*:final-xor 0*] [*:big-endian? #t*] [bigloo procedure]
`crc-string` *name str::bstring* [*:init 0*] [*:final-xor 0*] [*:big-endian? #t*] [bigloo procedure]
`crc-port` *name p::input-port* [*:init 0*] [*:final-xor 0*] [*:big-endian? #t*] [bigloo procedure]
`crc-mmap` *name m::mmap* [*init 0*] [*:final-xor 0*] [*big-endian? #t*] [bigloo procedure]
`crc-file` *name f::bstring* [*init 0*] [*:final-xor 0*] [*big-endian? #t*] [bigloo procedure]
Computes the CRC of the given object. *name* must be one of the provided CRC-algorithms. The optional parameter *init* can be used to initialize the CRC. The result of the CRC will be XORed with *final-xor*. The result will however be of the CRC's length. That is, even if *final-xor* is bigger then the CRC's length only the relevant bits will be used to perform the final XOR.

The result will be a number. Depending on the CRC this number can be a fixnum, an elong, or an llong.

The following example mimicks the UNIX `cksum` command:

```
(module cksum (main main))
(define (main args)
  (let loop ((sum (crc-file 'ieee-32 (cadr args)))
             (size (elong->fixnum (file-size (cadr args)))))
    (if (=fx size 0)
        (printf "~a ~a ~a\n"
                (bit-andllong #lxFFFFFFFF (elong->llong (bit-notelong sum)))
                (file-size (cadr args))
                (cadr args))
        (loop (crc-string 'ieee-32
                          (string (integer->char-ur (bit-and size #xFF)))
                          :init sum)
          (bit-rsh size 8)))))
```

In the following example we implement OpenPGP's CRC-24:

```
(define (openpgp-crc-24 str)
  (crc-string 'radix-64-24 str :init #xB704CE))
```

Be aware that many common CRCs use -1 as init value and invert the result. For compatibility with other implementations you might want to try one of the following alternatives:

```
(define (alt1 name obj) (crc name obj :init -1))
(define (alt2 name obj) (crc name obj :final-xor -1))
(define (alt3 name obj) (crc name obj :init -1 :final-xor -1))
```

Bigloo provides means to create additional CRCs: one can either simply provide a new polynomial or use Bigloo's low level functions.

`register-crc!` *name poly len* [bigloo procedure]

Adds the given CRC to Bigloo's list. Name can be of any type (`crc` will use `assoc` to find it in its list). The polynomial can be either a fixnum, an elong or an llong. *len* should give the CRCs size. The type of the polynomial and the given *len* must be consistent. On a 32 bit machine the following CRC registration would be invalid and yield undefined results:

```
(register-crc! 'invalid 1337 55)
```

As 55 is bigger than the fixnum's bit-size calling `crc` with this CRC will yield undefinde results.

`crc-long::long` *c::char crc::long poly::long len::long* [bigloo procedure]
`crc-elong::elong` *c::char crc::elong poly::elong len::long* [bigloo procedure]
`crc-llong::llong` *c::char crc::llong poly::llong len::long* [bigloo procedure]
`crc-long-le::long` *c::char crc::long poly::long len::long* [bigloo procedure]
`crc-elong-le::elong` *c::char crc::elong poly::elong len::long* [bigloo procedure]
`crc-llong-le::llong` *c::char crc::llong poly::llong len::long* [bigloo procedure]

These function perform a CRC operation on one byte. The previously described functions are based on these low level functions. The result of all the low level functions will return values that are not cut to the correct length. Usually a crc is done in a loop, and one needs to `bit-and` only when returning the result. Polynomials can be given with or without the high-order bit.

For instance we could implement `openpgp-crc24` as follows:

```
(define *openpgp-init* #xB704CE)
(define *radix-64-24-poly* #x864CFB)
(define (openpgp-crc-24 str)
  (let loop ((i 0)
             (crc *openpgp-init*))
    (if (=fx i (string-length str))
        (bit-and crc #xFFFFFF) ;; cut to correct length (24 bits)
        (loop (+fx i 1)
              (crc-long (string-ref str i) crc *radix-64-24-poly* 24)))))
```

`crc-polynomial-be->le` *len polynomial* [bigloo procedure]

Returns the little endian variant of a given polynomial.

5.11 Internet

This section presents the Bigloo function aimed at helping internet programming.

5.12 URLs

`url-parse` *url* [bigloo procedure]

The argument *url* can either be a string or an input-port. The function `url-parse` parses the url and returns four values:

- the protocol,
- the optional user info,
- the host name,
- the port number,
- the absolute path

Example

```
(multiple-value-bind (protocol uinfo host port abspath)
   (url-parse "http://www.inria.fr/sophia/teams/indes/index.html")
   (list protocol uinfo host port abspath))
      ⇒ ("http" #f "www.inria.fr" 80 "/sophia/teams/indes/index.html'')
(multiple-value-bind (protocol uinfo host port abspath)
   (url-parse "https://foo:bar@www.inria.fr/sophia/teams/indes/index.html")
   (list protocol uinfo))
      ⇒ ("https" "foo@bar")
```

`url-sans-protocol-parse` *url protocol* [bigloo procedure]

The argument *url* can either be a string or an input-port.

This function behaves as `url-parse` except it assumes that the protocol part of the url has already been extracted from the URI. It is explicitly provided using the *protocol* argument.

`http-url-parse` *url* [bigloo procedure]

The argument *url* can either be a string or an input-port. As `url-parse`, it returns four values.

This function parses URL found in HTTP GET responses.

`url-path-encode` *path* [bigloo procedure]

Encode a path that can be used in valid URL.

```
(url-path-encode "/tmp/foo") ⇒ "/tmp/foo"
(url-path-encode "/tmp/foo&bar") ⇒ "/tmp/foo%26bar"
(url-path-encode "http:///tmp/foo") ⇒ "http%3A//tmp/foo"
```

`url-encode` *url* [bigloo procedure]
`uri-encode` *url* [bigloo procedure]
`uri-encode-component` *url* [bigloo procedure]

Encode a URL by removing any illegal character.

```
(url-encode "http:///tmp/foo") ⇒ "http://tmp:80/foo"
(url-encode "http:///tmp/foo&bar") ⇒ "http://tmp:80/foo%26"
```

`url-decode` *url* [bigloo procedure]
`url-decode!` *url* [bigloo procedure]
`uri-decode` *url* [bigloo procedure]
`uri-decode!` *url* [bigloo procedure]
`uri-decode-component` *url* [bigloo procedure]
`uri-decode-component!` *url* [bigloo procedure]

Decode a URL. The function `url-decode!` may return its argument unmodified if no decoding is for the URL.

The variants `-component` treat do not escape URI reserved characters (i.e., #, /, ?, :, @, &, =, +, and $).

5.13 HTTP

`http` [*:in #f*] [*:out #f*] [*:socket #f*] [bigloo procedure]

[:protocol 'http] [:method 'get] [:timeout 0] [:proxy #f] [:host "localhost"] [:port 80] [:path "/"] [:login #f] [:authorization #f] [:username #f] [:password #f] [:http-version "HTTP/1.1"] [:content-type #f] [:connection "close"] [:header '((user-agent: "Mozilla/5.0"))] [:args '()] [:body #f]

Opens an HTTP connection. Returns a socket.

It is an error to specify a header twice. In particular, it is illegal to re-define keyworded arguments in the *:header* list. For instance, it is illegal to include in the `:header` actual list value a value for the `Connection` HTTP connection.

```
(define (wget url)

   (define (parser ip status-code header clen tenc)
      (if (not (and (>=fx status-code 200) (<=fx status-code 299)))
  (case status-code
     ((401)
      (raise (instantiate::&io-port-error
(proc 'open-input-file)
(msg "Cannot open URL, authentication required")
(obj url))))
     ((404)
      (raise (instantiate::&io-file-not-found-error
(proc 'open-input-file)
(msg "Cannot open URL")
(obj url))))
```

```
      (else
       (raise (instantiate::&io-port-error
(proc 'open-input-file)
(msg (format "Cannot open URL (~a)" status-code))
(obj url)))))
  (cond
     ((not (input-port? ip))
      (open-input-string ""))
     (clen
      (input-port-fill-barrier-set! ip (elong->fixnum clen))
      ip)
     (else
      ip))))

   (multiple-value-bind (protocol login host port abspath)
      (url-parse url)
      (let* ((sock (http :host host :port port :login login :path abspath))
     (ip (socket-input sock))
     (op (socket-output sock)))
 (with-handler
    (lambda (e)
       (if (isa? e &http-redirection)
                   (with-access::&http-redirection e (url)
      (wget url))
   (raise e)))
    (read-string (http-parse-response ip op parser))))))
```

The optional argument `args` is used for `post` method. The actual value should be a list of lists. Each of these sublists must have two values:

- the argument name
- the argument actual value

The argument name can be either a string which is the name of the argument or a list of two elements. In that case, the first element of these list is the argument name. The second element should be a string that denotes additional parameter.

Example:

```
(http :host "localhost" :port 8080 :method 'post
   :header '((enctype: "multipart/form-data"))
   :args `(("x" "foo") (("foo.scm" "filename=\"foo.scm\"\nContent-type: application/octet-stream" ,
   ...)
```

An http connection blocks until the connection is established. If the optional argument `timeout` is provided, the connection must be established before the specified time interval elapses. The timeout is expressed in microseconds.

`http-read-line` *input-port* [bigloo procedure]
`http-read-crlf` *input-port* [bigloo procedure]

Reads a line or an end-of-line of an HTTP response.

`http-parse-status-line` *input-port* [bigloo procedure]

Parses the status-line of an HTTP response. This returns a three values:

- The http version
- The status code
- the explanation phrase

`http-parse-header` *input-port output-port* [bigloo procedure]

Parses the whole header of an HTTP response. It returns multiple values which are:

- the whole header as an alist.
- the host given in the `host` header.
- the port given `host` field.
- the optional *content-length* header field.
- the optional *transfer-encoding* header field.
- the optional *authorization* header field.
- the optional *proxy-authorization* header field.
- the optional *connection* header field.

`http-parse-response` *input-port output-port procedure* [bigloo procedure]

Parses the whole response of an HTTP request. The argument *procedure* is invoked with five arguments:

- the input port to read the characters of the response,
- the status code,
- the header of the response,
- the content length,
- the type encoding.

`http-response-body->port` *input-port output-port* [bigloo procedure]

Parses an HTTP response and build an output port that delivers the characters of the content.

`http-chunks->procedure` *input-port* [bigloo procedure]

`http-chunks->port` *input-port* [bigloo procedure]

`http-send-chunks` *input-port output-port* [bigloo procedure]

6 Pattern Matching

Pattern matching is a key feature of most modern functional programming languages since it allows clean and secure code to be written. Internally, "pattern-matching forms" should be translated (compiled) into cascades of "elementary tests" where code is made as efficient as possible, avoiding redundant tests; Bigloo's "pattern matching compiler" provides this. The technique used is described in details in [QueinnecGeffroy92], and the code generated can be considered optimal[1] due to the way this "pattern compiler" was obtained.

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

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

6.1 Bigloo pattern matching facilities

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

`match-case` *key clause...* [bigloo syntax]

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

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

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

The equality predicate used is `eq?`.

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

The following syntax is also available:

`match-lambda` *clause...* [bigloo syntax]

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

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

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

6.2 The pattern language

The syntax for <pattern> is:

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

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

- ?- 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.
- #{foo (?- . ?-) (? integer?)} matches any structure or record foo whose first and second fields are respectively a pair and an integer. You can provide only the fields you want to test. The order is not relevant.

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

7 Fast search

This chapters details the Bigloo's API for fast string search algorithms.

7.1 Knuth, Morris, and Pratt

Bigloo supports an implementation of the *Knuth, Morris, and Pratt* algorithm on strings and memory mapped area, See Section 5.2.2 [Memory mapped area], page 67.

`kmp-table` *pattern* [bigloo procedure]

This function creates a *kmp*-table used for fast search.

`kmp-mmap` *kmp-table mmap offset* [bigloo procedure]
`kmp-string` *kmp-table string offset* [bigloo procedure]

This function searches the *pattern* described by *kmp-table* in the memory mapped area *mmap* (respec. in the *string*). The search starts at *offset*. If an occurrence is found, its position in the *mmap* is returned. Otherwise `-1` is returned.

For the sake of the example, here is a prototypal implementation of the Usenix command `grep`:

```
(define (main args)
   (cond
      ((null? (cdr args))
       (fprintf (current-error-port) "Usage: grep STRING [FILE]...")
       (exit 0))
      (else
       (let ((t (kmp-table (cadr args))))
  (for-each (lambda (f) (grep-file t f)) (cddr args))))))

(define (grep-file t file)
   (let* ((mm (open-mmap file read: #t write: #f))
  (ls (mmap-length mm)))
      (let loop ((o 0))
 (unless (>=fx o ls)
    (let ((n (kmp-mmap t mm o)))
       (when (>fx n 0)
  (print file ":" (mmap-line mm ls n))
  (loop (+fx n 1))))))
      (close-mmap mm)))

(define (mmap-line mm ls n)
   (let ((b 0)
 (e (elong->fixnum ls)))
      ;; beginning
      (let loop ((i n))
 (when (>fx i 0)
    (if (char=? (mmap-ref mm i) #\Newline)
(set! b (+fx i 1))
(loop (-fx i 1)))))
      ;; end
      (let loop ((i n))
 (when (<fx i ls)
    (if (char=? (mmap-ref mm i) #\Newline)
(set! e i)
(loop (+fx i 1)))))
      (mmap-substring mm b (- e b))))
```

7.2 Boyer - Moore

Bigloo supports an implementation of the *Boyer, Moore* algorithm on strings and memory mapped area, See Section 5.2.2 [Memory mapped area], page 67.

`bm-table` *pattern* [bigloo procedure]

This function creates a *bm*-table used for fast search.

`bm-mmap` *bm-table mmap offset* [bigloo procedure]
`bm-string` *bm-table string offset* [bigloo procedure]

This function searches the *pattern* described by *bm-table* in the memory mapped area *mmap* (respec. in the *string*). The search starts at *offset*. If an occurrence is found, its position in the *mmap* is returned. Otherwise `-1` is returned.

For the sake of the example, here is a prototypal implementation of the Usenix command `grep`:

```
(define (main args)
   (cond
      ((null? (cdr args))
       (fprintf (current-error-port) "Usage: grep STRING [FILE]...")
       (exit 0))
      (else
       (let ((t (bm-table (cadr args))))
  (for-each (lambda (f) (grep-file t f)) (cddr args))))))

(define (grep-file t file)
   (let* ((mm (open-mmap file read: #t write: #f))
  (ls (mmap-length mm)))
      (let loop ((o 0))
 (unless (>=fx o ls)
    (let ((n (bm-mmap t mm o)))
       (when (>fx n 0)
  (print file ":" (mmap-line mm ls n))
  (loop (+fx n 1))))))
      (close-mmap mm)))

(define (mmap-line mm ls n)
   (let ((b 0)
 (e (elong->fixnum ls)))
      ;; beginning
      (let loop ((i n))
 (when (>fx i 0)
    (if (char=? (mmap-ref mm i) #\Newline)
(set! b (+fx i 1))
(loop (-fx i 1)))))
      ;; end
      (let loop ((i n))
 (when (<fx i ls)
    (if (char=? (mmap-ref mm i) #\Newline)
(set! e i)
(loop (+fx i 1)))))
      (mmap-substring mm b (- e b))))
```

7.3 Boyer - Moore - Horspool

Bigloo supports an implementation of the *Boyer, Moore, Horspool* algorithm on strings and memory mapped area, See Section 5.2.2 [Memory mapped area], page 67.

`bmh-table` *pattern* [bigloo procedure]

This function creates a *bmh*-table used for fast search.

`bmh-mmap` *bm-table mmap offset* [bigloo procedure]

`bmh-string` *bm-table string offset* [bigloo procedure]

This function searches the *pattern* described by *bmh-table* in the memory mapped area *mmap* (respec. in the *string*). The search starts at *offset*. If an occurrence is found, its position in the *mmap* is returned. Otherwise `-1` is returned.

8 Structures and Records

Bigloo supports two kinds of enumerated types: the *structures* and the *records*. They offer similar facilities. Structures were pre-existing to records and they are maintained mainly for backward compatiblity. Recors are compliant with the Scheme request for implementation 9.

8.1 Structures

There is, in Bigloo, a new class of objects: structures, which are equivalent to C `struct`.

`define-struct` *name field...* [bigloo syntax]

This form defines a structure with name *name*, which is a symbol, having fields *field...* which are symbols or lists, each list being composed of a symbol and a default value. This form creates several functions: creator, predicate, accessor and assigner functions. The name of each function is built in the following way:

- Creator: `make-`*name*
- Predicate: *name*`?`
- Accessor: *name*`-`*field*
- Assigner: *name*`-`*field*`-set!`

Function `make-`*name* accepts an optional argument. If provided, all the slots of the created structures are filled with it. The creator named `name` accepts as many arguments as the number of slots of the structure. This function allocates a structure and fills each of its slots with its corresponding argument.

If a structure is created using `make-`*name* and no initialization value is provided, the slot default values (when provided) are used to initialize the new structure. For instance, the execution of the program:

```
(define-struct pt1 a b)
(define-struct pt2 (h 4) (g 6))

(make-pt1)
   ⇒ #{PT1 () ()}
(make-pt1 5)
   ⇒ #{PT1 5 5}
(make-pt2)
   ⇒ #{PT2 4 6}
(make-pt2 5)
   ⇒ #{PT2 5 5}
```

`struct?` *obj* [bigloo procedure]

Returns `#t` if and only if *obj* is a structure.

8.2 Records (SRFI-9)

Bigloo supports records has specified by SRFI-9. This section is a copy of the SRFI-9 specification by Richard Kelsey. This SRFI describes syntax for creating new data types, called record types. A predicate, constructor, and field accessors and modifiers are defined for each record type. Each new record type is distinct from all existing types, including other record types and Scheme's predefined types.

`define-record-type` *expression...* [syntax]

The syntax of a record-type definition is:

```
<record-type-definition> ↦ (define-record-type <type-name>
                                                (<constructor-name> <field-tag> ...)
                                                <predicate-name>
                                                <field-spec> ...)
<field-spec>             ↦ (<field-tag> <accessor-name>)
                           | (<field-tag> <accessor-name> <modifier-name>)
<field-tag>              ↦ <identifier>
<accessor-name>          ↦ <identifier>
<predicate-name>         ↦ <identifier>
<modifier-name>          ↦ <identifier>
<type-name>              ↦ <identifier>
```

`Define-record-type` is generative: each use creates a new record type that is distinct from all existing types, including other record types and Scheme's predefined types. Record-type definitions may only occur at top-level (there are two possible semantics for 'internal' record-type definitions, generative and nongenerative, and no consensus as to which is better).

an instance of `define-record-type` is equivalent to the following definitions:

- `<type-name>` is bound to a representation of the record type itself. Operations on record types, such as defining print methods, reflection, etc. are left to other SRFIs.
- `<constructor-name>` is bound to a procedure that takes as many arguments as the re are `<field-tag>`s in the `(<constructor-name> ...)` subform and returns a new `<type-name>` record. Fields whose tags are listed with `<constructor-name>` have the corresponding argument as their initial value. The initial values of all other fields are unspecified.
- `<predicate-name>` is a predicate that returns `#t` when given a value returned by `<constructor-name>` and `#f` for everything else.
- Each `<accessor-name>` is a procedure that takes a record of type `<type-name>` and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each `<modifier-name>` is a procedure that takes a record of type `<type-name>` and a value which becomes the new value of the corresponding field; an unspecified value is returned. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

Records are disjoint from the types listed in Section 4.2 of R5RS.

Seting the value of any of these identifiers has no effect on the behavior of any of their original values.

The following

```
(define-record-type pare
    (kons x y)
    pare?
    (x kar set-kar!)
    (y kdr))
```

defines `kons` to be a constructor, `kar` and `kdr` to be accessors, `set-kar!` to be a modifier, and `pare?` to be a predicate for `pare`s.

```
(pare? (kons 1 2))        ⇒ #t
```

```
(pare? (cons 1 2))        ⇒ #f
(kar (kons 1 2))          ⇒ 1
(kdr (kons 1 2))          ⇒ 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k))                ⇒ 3
```

9 Object System

Bigloo's object system is designed to be as simple as possible and belongs to the CLOS [Bobrow et al. 88] object system family in that it uses *classes*, *generic functions* and *methods*. Its design has been strongly influenced by C. Queinnec's MEROON [Queinnec93] It does not include any meta object protocol.

9.1 Class declaration

Classes are defined in a module declaration. A class declaration can take place in a compiled or interpreted module. If a class declaration takes place in a static module clause (see Section Section 2.2 [Module Declaration], page 7) its scope is limited to the current module but if it takes place in an export module clause, its scope is extended to all modules that import the current module. The syntax of a class declaration is:

`class` *ident field . . .* [bigloo module clause]

```
<class> ↦ (class <ident> <constructor>? <field>+)
     | (final-class <ident> <constructor>? <field>+)
     | (wide-class <ident> <constructor>? <field>+)
     | (abstract-class <ident> <constructor>? <field>+)
<constructor> ↦ ( <expr> )
<field> ↦ <ident>
     | (<ident> <field-prop>)
<field-prop> ↦ read-only
     | (get <bigloo-exp>)
     | (set <bigloo-exp>)
     | (default <bigloo-exp>)
     | (info <bigloo-exp>)
```

A class is a Bigloo type (see Section Section 26.1.6.1 [Atomic types], page 237) and the class identifier is extracted from the <ident> of the class definition. If <ident> is also a <typed-ident>, the type part of this identifier denote the super-class of the class. If <ident> is a <IEEE-ident>, the super-class of the class is the root of the inheritance tree, the `object` class. This `object` class is the only pre-existing class.

Final classes can only be sub-classed by *wide classes*. *Wide classes* (only for compiled modules) can only inherit from *final classes*. *abstract classes* can't be instantiated.

Wide-classes cannot be defined within the interpreter.

The optional constructor is an expression that must evaluate to a one argument function. This function is automatically invoked each time a new class instance is created. The constructor is passed the fresh instance. If a class has not defined a constructor the super class' constructors are searched. The first constructor found is invoked. A constructor may be a generic function with a method specified for one or more classes.

A class field may be a *typed class field* which is achieved by using a <typed-ident> instead of a <IEEE-ident> for the <ident> value.

Field marked with `read-only` declaration are immutables.

`Default` declarations allow default field values.

For the means of an example, the traditional points and colored points can be defined as:

```
(module example
   (static (abstract-class pt)
           (class point::pt
              x::double
              y::double)
           (class point-C::point
              (color::string read-only))))
```

We illustrate final and wide classes by the example:

```
(module example
   (export (final-class person
               (name::string (default "Jones"))
               (sex read-only)
               children::pair-nil)
           (wide-class married-person::person
               mate::person)))
```

Fields may be virtual. A field is virtual as soon as its declaration contain a `get` attribute. Virtual fields have no physical implementation within the instance. When defining a virtual field, the class declaration implements a *getter* and a *setter* if that field is not a read only field. Access to the virtual field will rely on invocation of the user *getter* and user *setter*. For instance:

```
(module example
   (static (class complex
              mag::double
              angle::double
              (real::double (get (lambda (p)
                                    (with-access::complex p (mag angle)
                                       (* mag (cos angle)))))
                            read-only)
              (imag::double (get (lambda (p)
                                    (with-access::complex p (mag angle)
                                       (* mag (sin angle)))))
                            read-only))))

(let ((p (instantiate::complex (mag 1.0) (angle 2.18))))
   (with-access::complex p (real imag)
      (print "real: " real)
      (print "imag: " imag)))
```

Virtual fields cannot be associated default values. If a virtual field is not provided with a setter it must be annotated as read only.

`Info` declarations allow arbitrary user information field values. This value can be retrieved by introspection, by the means of the `class-field-info` introspection function.

For the means of an example, with add to information to the slot of the point class.

```
(module example
   (static (class point
              (x::double (info '(range 0.0 10.0)))
              (y::double (info '(range -1.0 1.0))))))
```

9.2 Creating and accessing objects

Objects and classes are created and manipulated via library functions and forms created automatically by Bigloo when a new class is defined.

`isa?` *obj class* [bigloo procedure]

This function returns `#t` if *obj* is an instance of *class* or an instance of a sub-class of *class*, otherwise, it returns `#f`.

`instantiate::`*`class`* (*ident value*)... [bigloo syntax]

This forms allocates object of class *class* and fills the fields with values found in the list of parameters (note that field are explicitly named and that there is no ordering for field naming). Field values which are not provided in the parameter list must have been declared with a `default` value which is used to initialize the corresponding field.

For instance:

```
(module example
   (export
      (class point (x (default 0)))
      (class point2d::point y)))

(instantiate::point (x 0) (y 0))
(instantiate::point (y 0))
(instantiate::point (x 0))
  ⇒ Error because y has no default value
```

`class-nil` *class* [bigloo procedure]

This function returns the NIL pre-existing class instance. This instance plays the role of `void *` in C or `null` in Java. The value of each field is unspecified but correct with respect to the Bigloo type system. Each call to *`class`*`-nil` returns the same object (in the sense of `eq?`).

```
(module example
   (export
      (class point x)
      (class point2d::point y)))

(eq? (class-nil point) (class-nil point))
  ⇒ #t
(eq? (class-nil point) (class-nil point2d))
  ⇒ #f
```

`with-access::`*`class`* *obj* (*binding*...) *body* [bigloo syntax]

A reference to any of the variables defined in as a *binding* is replaced by the appropriate field access form. This is true for both reference and assignment. A *binding* is either a symbol or a list of two symbols. In the first place, it denotes a field. In the second case, it denotes an aliases field.

For instance:

```
(with-access::point p (x (y1 y))
   (with-access::point p2 (y)
      (set! x (- x))
      (set! y1 (- y1 y))))
```

`-> var field` [bigloo syntax]

Class instances can be accesses using the `->` special form. The the first argument must be the identifier of a local typed variable, otherwise an error is raised. The form `->` can be used to get or set value of an instance field. For instance:

```
(define (example p1::point p2::point)
   (set! (-> p1 x) (- (-> p1 x)))
   (set! (-> p1 y) (- (-> p1 y) (-> p2 y))))
```

This is equivalent to:

```
(define (example p1::point p2::point)
   (with-access::point p1 (x (y1 y))
      (with-access::point p2 (y)
         (set! x (- x))
         (set! y1 (- y1 y)))))
```

`co-instantiate` ((*var value*) ...) *body* [bigloo syntax]

This form is only available from compiled modules. In other words, it is not available from the interpreter. It permits the creation of recursive instances. It is specially useful for creating instances for which class declarations contain cyclic type references (for instance a class `c1` for a which a field is declared of class `c2` and a class `c2` for which a class is declared of type `c1`). The syntax of a `co-instantiate` form is similar to a `let` form. However the only legal *values* are `instantiate` forms. The variables introduced in the binding of a `co-instantiate` form are bound in *body*. In addition, they are *partially* bound in the *values* expressions. In a *value* position, a variable *var* can only be used to set the value of a field of an instantiated class. It cannot be used in any calculus. Example:

```
(module obj-example
   (export (class c1 a b o2::c2)
           (class c2 x y o1::c1)))

(co-instantiate ((o1 (instantiate::c1
                        (a 10)
                        (b 20)
                        (o2 o2)))
                 (o2 (instantiate::c2
                        (x 10)
                        (y 20)
                        (o1 o1))))
   (with-access::c1 o1 (o2)
      (with-access::c2 o2 (x y)
         (+ x y))))
   ⇒ 30
```

`duplicate::`*class obj* (*ident value*)... [bigloo syntax]

This forms allocates an instance of class *class*. The field values of the new object are picked up from the field values of the *old* object unless they are explicitly given in the parameter list.

For instance:

```
(with-access::point old (x)
   (instantiate::point
      (x x)
      (y 10)))
```

is equivalent to:

```
(duplicate::point old (y 10))
```

The duplicated instance might be a subclass of duplicated instance. Example:

```
(module foo
   (export (class point x y)
   (class point3::point z)
   (class point4::point3 t))
   (main main))

(define (main x)
   (let ((p (instantiate::point (x 1) (y 2))))
      (print (duplicate::point4 p (z 6) (t 55)))))
```

9.3 Generic functions

A generic function is a bag of specific functions known as methods. When invoked on a Bigloo object, a generic function determines the class of the discriminating variable (corresponding to the first argument of the generic function) and invokes the appropriate method. Generic functions implement single inheritance and each is defined using the `define-generic` Bigloo syntax.

`define-generic` (*name arg...*) *default-body* [bigloo syntax]

A generic function can be defined with a default body which will be evaluated if no method can be found for the discriminating variable. The default default-body signals an error.

As an example, here is a possible definition of the `object-display` generic function:

```
(define-generic (object-display obj::object . op)
   (let ((port (if (pair? op)
                   (car op)
                   (current-output-port))))
      (display "#\|" port)
      (display (class-name (object-class obj)) port)
      (display "\|" port)))
```

Methods can be defined to specialize a generic function and such methods must have a compatible variable list. That is, the first argument of the method must be a sub-type (i.e. belong to a sub-class) of the first argument of the generic function. Other formal parameters must be of same types. Moreover, the result type of the method must be a sub-type of the result of the generic function.

`define-method` (*name arg...*) *body* [bigloo syntax]
`call-next-method` [bigloo syntax]

If there is no appropriate method, an error is signaled.

Methods can use the form `(call-next-method)` to invoke the method that would have been called if not present. The `(call-next-method)` cannot be used out of method definition. example:

```
(define-method (object-display p::person . op)
   (let ((port (if (pair? op)
                   (car op)
                   (current-output-port))))
      (fprint port "firstname : " (-> p fname))
```

```
(fprint port "name      : " (-> p name))
(fprint port "sex       : " (-> p sex))
p))
```

9.4 Widening and shrinking

Bigloo introduces a new kind of inheritance: *widening*. This allows an object to be temporarily *widened* (that is transformed into an object of another class, a *wide-class*) and then *shrink-ed* (that is reshaped to its original class). This mechanism is very useful for implementing short-term data storage. For instance, Bigloo compilation passes are implemented using the *widening/shrinking* mechanism. On entry to a pass, objects are widened with the specific pass fields and, on exit from a pass, objects are shrunk in order to forget the information related to this pass.

Only instances of *final classes* can be widened and objects can only be widened in order to become instances of *wide classes*. Widening is performed by the `widen!` syntax:

`widen!::`*`wide-class`* *obj* (*id value*) *...* [bigloo syntax]

The object *obj* is widened to be instance of the wide class *wide-class*. Fields values are either picked up from the parameter list of the `widen!` form or from the default values in the declaration of the wide class.

Objects are shrunk using the `shrink!` syntax:

`shrink!` *obj* [bigloo syntax]

Here is a first example:

```
(module example
   (static (final-class point
              (x (default 0))
              (y (default 0)))
           (wide-class named-point::point name)))

(define *point* (instantiate::point))
```

Two classes have been declared and an instance `*point*` of `point` has been allocated. For now, `*point*` is an instance of `point` but not an instance of `named-point` and this can be checked by:

```
(print (isa? *point* named))          ↦ #t
(print (isa? *point* named-point))    ↦ #f
```

Now, we *widen* `*point*`...

```
(let ((n-point (widen!::named-point *point*
                  (name "orig"))))
```

And we check that now, `n-point` is an instance of `named-point`. Since `named-point` is a subclass of `point`, `n-point` still is an instance of `point`.

```
(print (isa? n-point named-point))  ↦ #t
(print (isa? n-point named))        ↦ #t
```

Widening affects the objects themselves. It does not operate any copy operation. Hence, `*point*` and `n-point` are `eq?`.

```
(print (eq? n-point *point*))   ↦ #t
```

To end this example, we *shrink* `n-point` and check its class.

```
(shrink! n-point)
```

```
(print (isa? *point* named-point))) ↦ #f
```

Here is a more complex example:

We illustrate widening and shrinking using our "wedding simulator". First let us define three classes, `person` (for man and woman), `married-woman` and `married-man`:

```
(module wedding
   (static (final-class person
               name::string
               fname::string
               (sex::symbol read-only))
           (wide-class married-man::person
               mate::person)
           (wide-class married-woman::person
               maiden-name::string
               mate::person)))
```

As we can see people are allowed to change their name but not their sex.

The identity of a person can be printed as

```
(define-method (object-display p::person . op)
   (with-access::person p (name fname sex)
      (print "firstname : " fname)
      (print "name      : " name)
      (print "sex       : " sex)
      p))
```

A married woman's identity is printed by (we suppose an equivalent method definition for married-man)

```
(define-method (object-display p::married-woman . op)
   (with-access::married-woman p (name fname sex mate)
      (call-next-method)
      (print "married to: " mate)
      p))
```

We create a person with the `birth` function:

```
(define (birth name::string fname::string sex)
   [assert (sex) (memq sex '(male female))]
   (instantiate::person
      (name name)
      (fname fname)
      (sex sex)))
```

We celebrate a wedding using the `get-married!` function:

```
(define (get-married! woman::person man::person)
   (if (not (and (eq? (-> woman sex) 'female)
                 (eq? (-> man sex) 'male)))
       (error "get-married"
              "Illegal wedding"
              (cons woman man))
       (let* ((mname (-> woman name))
              (wife  (widen!::married-woman woman
                        (maiden-name mname)
                        (mate man))))
          (person-name-set! wife (-> man name))
          (widen!::married-man man
             (mate woman)))))
```

We can check if two people are married by

```
(define (couple? woman::person man::person)
```

```
(and (isa? woman married-woman)
     (isa? man married-man)
     (eq? (with-access::married-woman woman (mate) mate) man)
     (eq? (with-access::married-man man (mate) mate) woman)))
```

Now let us study the life a `Junior Jones` and `Pamela Smith`. Once upon a time...

```
(define *junior* (birth "Jones" "Junior" 'male))
(define *pamela* (birth "Smith" "Pamela" 'female))
```

Later on, they met each other and ... they got married:

```
(define *old-boy-junior* *junior*)
(define *old-girl-pamela* *pamela*)
(get-married! *pamela* *junior*)
```

This union can be checked:

```
(couple? *pamela* *junior*)
   ⇒ #t
```

We can look at the new identity of `*pamela*`

```
(print *pamela*)
   ⊣ name      : Jones
     firstname : Pamela
     sex       : FEMALE
     married to: Junior Jones
```

But `*pamela*` and `*junior*` still are the same persons:

```
(print (eq? *old-boy-junior* *junior*))   ⇒ #t
(print (eq? *old-girl-pamela* *pamela*))  ⇒ #t
```

Unfortunately all days are not happy days. After having been married `*pamela*` and `*junior*` have divorced:

```
(define (divorce! woman::person man::person)
   (if (not (couple? woman man))
       (error "divorce!"
              "Illegal divorce"
              (cons woman man))
       (with-access::married-woman woman (maiden-name)
          (begin
             (shrink! woman)
             (set! (-> woman name) maiden-name))
          (shrink! man))))

(divorce! *pamela* *junior*)
```

We can look at the new identity of `*pamela*`

```
(print *pamela*)
   ⊣ name      : Smith
     firstname : Pamela
     sex       : FEMALE
```

And `*pamela*` and `*junior*` still are the same persons:

```
(print (eq? *old-boy-junior* *junior*))   ⇒ #t
(print (eq? *old-girl-pamela* *pamela*))  ⇒ #t
```

9.5 Object library

9.5.1 Classes handling

No type denotes Bigloo's classes. These objects are handled by the following library functions:

`find-class` *symbol* [bigloo procedure]
Returns, if any, the class named *symbol*.

`class?` *obj* [bigloo procedure]
Returns `#t` if and only if *obj* is a class.

`class-super` *class* [bigloo procedure]
Returns the *super-class* of *class*.

`class-subclasses` *class* [bigloo procedure]
Returns the *subclasses* of *class*.

`class-name` *class* [bigloo procedure]
Returns the name (a symbol) of *class*.

`object-constructor` *class* [bigloo procedure]
Returns *class*'s constructor.

`object-class` *object* [bigloo procedure]
Returns the class that *object* belongs to.

9.5.2 Object handling

`wide-object?` *object* [bigloo procedure]
Returns `#t` if *object* is a wide object otherwise it returns `#f`.

`object-display` *object* [*port*] [bigloo generic]
This generic function is invoked by `display` to display objects.

`object-write` *object* [*port*] [bigloo generic]
This generic function is invoked by `write` to write objects.

`object->struct` *object* [bigloo generic]
`struct->object` *struct* [bigloo procedure]
These functions converts objects into Scheme structures and vice-versa.

`object-equal?` *object obj* [bigloo generic]
This generic function is invoked by `equal?` when the first argument is an instance of `object`.

`object-hashnumber` *object* [bigloo generic]
This generic function returns an hash number of *object*.

`is-a?` *obj class* [bigloo procedure]
Returns `#t` if *obj* belongs to *class* otherwise it returns `#f`.

9.6 Object serialization

Objects can be *serialized* and *un-serialized* using the regular `string->obj` and `obj->string` functions. Objects can be stored on disk and restored from disk by the use of the `output-obj` and `input-obj` functions.

In addition to this standard serialization mechanism, custom object serializers and un-serializers can be specified by the means of the `register-class-serialization!` function (see Section Section 5.3 [Serialization], page 70.

9.7 Equality

Two objects can be compared with the `equal?` function. Two object are equal if and only if they belong to a same class, all their field values are equal and all their super class's field values are equal.

9.8 Introspection

Bigloo provides the programmer with some object introspection facilities. See section see Section 9.5 [Object library], page 124, for information on classes and objects handling. Introspection facilities are, by default, available for all classes. However, in order to shrink the code size generation, it may be useful to disable class introspection. This decision can be taken on a per class basis (i.e., one class may be provided with introspection facilities while another one is not). The compiler option `-fno-reflection` (see Chapter Chapter 31 [Compiler Description], page 275) prevents the compiler to generate the code required for introspecting the classes defined in the compiled module.

`class-fields` *class* [bigloo procedure]
: Returns the a description of the fields of *class*. This description is a list of field descriptions where each field description can be accessed by the means of the following library functions. The fields are those *directly* defined in *class*. That is `class-fields` does not return fields defined in super classes of *class*.

`class-all-fields` *class* [bigloo procedure]
: Returns the a description of the fields of *class*. This description is a list of field descriptions where each field description can be accessed by the means of the following library functions. By contrast with `class-fields`, this function returns fields that are also defined in the super classes of *class*. in th

`find-class-field` *class symbol* [bigloo procedure]
: Returns the field named *symbol* from class *class*. Returns `#f` is such a field does not exist.

`class-field?` *obj* [bigloo procedure]
: Returns #t if *obj* is a class field descriptor. Otherwise returns #f.

`class-field-name` *field* [bigloo procedure]
: Returns the name of the *field*. The name is a symbol.

`class-field-accessor` *field* [bigloo procedure]
: Returns a procedure of one argument. Applying this function to an object returns the value of the field described by *field*.

`class-field-mutable?` *field* [bigloo procedure]
Returns `#t` if the described field is mutable and `#f` otherwise.

`class-field-mutator` *field* [bigloo procedure]
Returns a procedure of two arguments. Applying this function to an object changes the value of the field described by *field*. It is an error to apply `class-field-mutator` to an immutable field.

`class-field-info` *field* [bigloo procedure]
Returns the information associated to *field* (this the class declaration `info` attribute).

For means of an example, here is a possible implementation of the `equal?` test for objects:

```
(define (object-equal? obj1 obj2)
   (define (class-field-equal? fd)
      (let ((get-value (class-field-accessor fd)))
          (equal? (get-value obj1) (get-value obj2))))
   (let ((class1 (object-class obj1))
         (class2 (object-class obj2)))
      (cond
         ((not (eq? class1 class2))
          #f)
         (else
          (let loop ((fields (class-fields class1))
                     (class  class1))
             (cond
                ((null? fields)
                 (let ((super (class-super class)))
                    (if (class? super)
                        (loop (class-fields super)
                              super)
                        #t)))
                ((class-field-equal? (car fields))
                 (loop (cdr fields) class))
                (else
                 #f)))))))
```

`class-creator` *class* [bigloo procedure]
Returns the creator for *class*. The creator is a function for which the arity depends on the number of slots the class provides (see Section see Section 9.2 [Creating and accessing objects], page 119).

When an instance is allocated by the means of the `class-creator`, as for direct instantiation, the class constructor is *automatically* invoked. Example:

```
(module foo
   (main main)
   (static (class c1 (c1-constructor))))

(define c1-constructor
   (let ((count 0))
      (lambda (inst)
 (set! count (+ 1 count))
 (print "creating instance: " count)
 inst)))

(define (main argv)
   (let ((o1 (instantiate::c1))
```

```
(o2 (instantiate::c1))
(o3 ((class-creator c1))))
     'done))
  ⊣ creating instance: 1
    creating instance: 2
    creating instance: 3
```

`class-predicate` *class* [bigloo procedure]

Returns the predicate for *class*. This predicate returns `#t` when applied to object of type *class*. It returns `#f` otherwise.

10 Regular parsing

Programming languages have poor reading libraries since the lexical information that can be specified is directly tied to the structure of the language. For example, in C it's hard to read a rational number because there is no type rational. Programs have been written to circumvent this problem: Lex [Lesk75], for example, is one of them. We choose to incorporate in Bigloo a set of new functions to assist in such parsing. The syntax for regular grammar (also known as regular analyser) of Bigloo 2.0 (the one described in this document) is not compatible with former Bigloo versions.

10.1 A new way of reading

There is only one way in Bigloo to read text, *regular reading*, which is done by the new form:

`read/rp` *regular-grammar input-port* [bigloo procedure]

The first argument is a regular grammar (also known as regular analyser) and the second a Scheme port. This way of reading is almost the same as the Lex's one. The reader tries to match the longest input, from the stream pointed to by *input-port*, with one of several regular expressions contained in *regular-grammar*. If many rules match, the reader takes the first one defined in the grammar. When the regular rule has been found the corresponding Scheme expression is evaluated.

remark: The traditional `read` Scheme function is implemented as:

```
(define-inline (read port)
   (read/rp scheme-grammar port))
```

10.2 The syntax of the regular grammar

A regular grammar is built by the means of the form `regular-grammar`:

`regular-grammar` (*binding ...*) *rule ...* [bigloo syntax]

The *binding* and *rule* are defined by the following grammar:

```
<binding>  ↦ (<variable> <re>)
             | <option>
<option>   ↦ <variable>
<rule>     ↦ <define>
             | (<cre> <s-expression> <s-expression> ...)
             | (else <s-expression> <s-expression> ...)
<define>   ↦ (define <s-expression>)
<cre>      ↦ <re>
             | (context <symbol> <re>)
             | (when <s-expr> <re>)
             | (bol <re>)
             | (eol <re>)
             | (bof <re>)
             | (eof <re>)
<re>       ↦ <variable>
             | <char>
             | <string>
             | (: <re> ...)
             | (or <re> ...)
             | (* <re>)
```

```
                | (+ <re>)
                | (? <re>)
                | (= <integer> <re>)
                | (>= <integer> <re>)
                | (** <integer> <integer> <re>)
                | (... <integer> <re>)
                | (uncase <re>)
                | (in <cset> ...)
                | (out <cset> ...)
                | (and <cset> <cset>)
                | (but <cset> <cset>)
                | (posix <string>)
<variable> ↦ <symbol>
<cset>     ↦ <string>
                | <char>
                | (<string>)
                | (<char> <char>)
```

Here is a description of each construction.

`(context <symbol> <re>)`
: This allows us to *protect* an expression. A *protected* expression matches (or accepts) a word only if the grammar has been set to the corresponding context. See Section 10.3 [The Semantics Actions], page 133, for more details.

`(when <s-expr> <re>)`
: This allows us to *protect* an expression. A *protected* expression matches (or accepts) a word only if the evaluation of `<s-expr>` is `#t`. For instance,

```
(define *g*
   (let ((armed #f))
      (regular-grammar ()
       ((when (not armed) (: "#!" (+ (or #\/ alpha))))
        (set! armed #t)
        (print "start [" (the-string) "]")
        (ignore))
       ((+ (in #\Space #\Tab))
        (ignore))
       (else
        (the-failure)))))

(define (main argv)
   (let ((port (open-input-string "#!/bin/sh #!/bin/zsh")))
      (print (read/rp *g* port))))
```

`(bol <re>)`
: Matches `<re>` at the beginning of line.

`(eol <re>)`
: Matches `<re>` at the end of line.

`(bof <re>)`
: Matches `<re>` at the beginning of file.

`(eof <re>)`
: Matches `<re>` at the end of file.

`<variable>`

This is the name of a variable bound by a `<binding>` construction. In addition to user defined variables, some already exist. These are:

```
all    ≡ (out #\Newline)
lower  ≡ (in ("az"))
upper  ≡ (in ("AZ"))
alpha  ≡ (or lower upper)
digit  ≡ (in ("09"))
xdigit ≡ (uncase (in ("af09")))
alnum  ≡ (uncase (in ("az09")))
punct  ≡ (in ".,;!?")
blank  ≡ (in #" \t\n")
space  ≡ #\Space
```

It is a error to reference a variable that it is not bound by a `<binding>`. Defining a variable that already exists is acceptable and causes the former variable definition to be erased. Here is an example of a grammar that binds two variables, one called '`ident`' and one called '`number`'. These two variables are used within the grammar to match identifiers and numbers.

```
(regular-grammar ((ident  (: alpha (* alnum)))
                  (number (+ digit)))
   (ident  (cons 'ident (the-string)))
   (number (cons 'number (the-string)))
   (else   (cons 'else (the-failure))))
```

`<char>` The regular language described by one unique character. Here is an example of a grammar that accepts either the character `#\a` or the character `#\b`:

```
(regular-grammar ()
   (#\a (cons 'a (the-string)))
   (#\b (cons 'b (the-string)))
   (else (cons 'else (the-failure))))
```

`<string>` This simple form of regular expression denotes the language represented by the string. For instance the regular expression `"Bigloo"` matches only the string composed of `#\B #\i #\g #\l #\o #\o`. The regular expression `".*["` matches the string `#\. #\* #\[`.

`(: <re> ...)`

This form constructs sequence of regular expression. That is a form `<re1> <re2> ... <ren>` matches the language construction by concatenation of the language described by `<re1>`, `<re2>`, `<ren>`. Thus, `(: "x" all "y")` matches all words of three letters, started by character the `#\x` and ended with the character `#\y`.

`(or <re> ...)`

This construction denotes conditions. The language described by `(or re1 re2)` accepts words accepted by either `re1` or `re2`.

`(* <re>)` This is the Kleene operator, the language described by `(* <re>)` is the language containing, 0 or more occurrences of `<re>`. Thus, the language described by `(* "abc")` accepts the empty word and any word composed by a repetition of the `abc` (`abc`, `abcabc`, `abcabcabc`, ...).

`(+ <re>)` This expression described non empty repetitions. The form `(+ re)` is equivalent to `(: re (* re))`. Thus, `(+ "abc")` matches the words `abc`, `abcabc`, etc.

`(? <re>)` This expression described one or zero occurrence. Thus, `(? "abc")` matches the empty word or the words `abc`.

`(= <integer> <re>)`
This expression described a fix number of repetitions. The form `(= num re)` is equivalent to `(: re re ... re)`. Thus, the expression `(= 3 "abc")` matches the only word `abcabcabc`. In order to avoid code size explosion when compiling, `<integer>` must be smaller than an arbitrary constant. In the current version that value is `81`.

`(>= <integer> <re>)`
The language described by the expression `(>= int re)` accepts word that are, at least, `int` repetitions of `re`. For instance, `(>= 10 #\a)`, accepts words compound of, at least, 10 times the character `#\a`. In order to avoid code size explosion when compiling, `<integer>` must be smaller than an arbitrary constant. In the current version that value is `81`.

`(** <integer> <integer> <re>)`
The language described by the expression `(** min max re)` accepts word that are repetitions of `re`; the number of repetition is in the range `min`, `max`. For instance, `(** 10 20 #\a)`. In order to avoid code size explosion when compiling, `<integer>` must be smaller than an arbitrary constant. In the current version that value is `81`.

`(... <integer> <re>)`
The subexpression `<re>` has to be a sequence of characters. Sequences are build by the operator `:` or by string literals. The language described by `(... int re)`, denotes, the first letter of `re`, or the two first letters of `re`, or the three first letters of `re` or the `int` first letters of `re`. Thus, `(... 3 "begin")` is equivalent to `(or "b" "be" "beg")`.

`(uncase <re>)`
The subexpression `<re>` has to be a sequence construction. The language described by `(uncase re)` is the same as `re` where letters may be upper case or lower case. For instance, `(uncase "begin")`, accepts the words `"begin"`, `"beGin"`, `"BEGIN"`, `"BegiN"`, etc.

`(in <cset> ...)`
Denotes union of characters. Characters may be described individually such as in `(in #\a #\b #\c #\d)`. They may be described by strings. The expression `(in "abcd")` is equivalent to `(in #\a #\b #\c #\d)`. Characters may also be described using a range notation that is a list of two characters. The expression `(in (#\a #\d))` is equivalent to `(in #\a #\b #\c #\d)`. The Ranges may be expresses using lists of string. The expression `(in ("ad"))` is equivalent to `(in #\a #\b #\c #\d)`.

`(out <cset> ...)`
: The language described by `(out cset ...)` is opposite to the one described by `(in cset ...)`. For instance, `(out ("azAZ") (#\0 #\9))` accepts all words of one character that are neither letters nor digits. One should not that if the character numbered zero may be used inside regular grammar, the `out` construction never matches it. Thus to write a rule that, for instances, matches every character but `#\Newline` including the character zero, one should write:

```
(or (out #\Newline) #a000)
```

`(and <cset> <cset>)`
: The language described by `(and cset1 cset2)` accepts words made of characters that are in both `cset1` and `cset2`.

`(but <cset> <cset>)`
: The language described by `(but cset1 cset2)` accepts words made of characters of `cset1` that are not member of `cset2`.

`(posix <string>)`
: The expression `(posix string)` allows one to use Posix string notation for regular expressions. So, for example, the following two expressions are equivalent:

```
(posix "[az]+|x*|y{3,5}")

(or (+ (in ("az"))) (* "x") (** 3 5 "y"))
```

`string-case` *string rule ...* [bigloo syntax]
: This form dispatches on strings. it opens an input on `string` a read into it according to the regular grammar defined by the `binding` and `rule`. Example:

```
(define (suffix string)
   (string-case string
      ((: (* all) ".")
       (ignore))
      ((+ (out #\.))
       (the-string))
      (else
       "")))
```

10.3 The semantics actions

The semantics actions are regular Scheme expressions. These expressions appear in an environment where some "extra procedures" are defined. These procedures are:

`the-port` [bigloo rgc procedure]
: Returns the input port currently in used.

`the-length` [bigloo rgc procedure]
: Get the length of the biggest matching string.

`the-string` [bigloo rgc procedure]
: Get a copy of the last matching string. The function `the-string` returns a fresh copy of the matching each time it is called. In consequence,

```
(let ((l1 (the-string)) (l2 (the-string)))
```

```
(eq? l1 l2))
⇒ #f
```

`the-substring` *start len* [bigloo rgc procedure]

Get a copy of a substring of the last matching string. If the *len* is negative, it is subtracted to the whole match length. Here is an example of a rule extracting a part of a match:

```
(regular-grammar ()
   ((: #\" (* (out #\")) #\")
    (the-substring 1 (-fx (the-length) 1))))
```

Which can also be written:

```
(regular-grammar ()
   ((: #\" (* (out #\")) #\")
    (the-substring 1 -1)))
```

`the-character` [bigloo rgc procedure]
`the-byte` [bigloo rgc procedure]

Returns the first character of a match (respectively, the first byte).

`the-byte-ref` *n* [bigloo rgc procedure]

Returns the *n*-th bytes of the matching string.

`the-symbol` [bigloo rgc procedure]
`the-downcase-symbol` [bigloo rgc procedure]
`the-upcase-symbol` [bigloo rgc procedure]
`the-subsymbol` *start length* [bigloo rgc procedure]

Convert the last matching string into a symbol. The function `the-subsymbol` obeys the same rules as `the-substring`.

`the-keyword` [bigloo rgc procedure]
`the-downcase-keyword` [bigloo rgc procedure]
`the-upcase-keyword` [bigloo rgc procedure]

Convert the last matching string into a keyword.

`the-fixnum` [bigloo rgc procedure]

The conversion of the last matching string to fixnum.

`the-flonum` [bigloo rgc procedure]

The conversion of the last matching string to flonum.

`the-failure` [bigloo rgc procedure]

Returns the first char that the grammar can't match or the end of file object.

`ignore` [bigloo rgc procedure]

Ignore the parsing, keep reading. It's better to use `(ignore)` rather than an expression like `(read/rp` *grammar port*`)` in semantics actions since the `(ignore)` call will be done in a tail recursive way. For instance,

```
(let ((g (regular-grammar ()
            (")"
             '())
            ("("
```

```
                (let* ((car (ignore))
                       (cdr (ignore)))
                   (cons car cdr)))
                ((+ (out "()"))
                 (the-string))))
          (p (open-input-string "(foo(bar(gee)))")))
      (read/rp g p))
      ⇒ ("foo" ("bar" ("gee")))
```

`rgc-context` [*context*] [bigloo rgc procedure]

If no *context* is provide, this procedure reset the reader context state. That is the reader is in no context. With one argument, `context` set the reader in the context *context*. For instance,

```
(let ((g (regular-grammar ()
           ((context foo "foo") (print 'foo-bis))
           ("foo" (rgc-context 'foo) (print 'foo) (ignore))
           (else 'done)))
      (p (open-input-string "foofoo")))
  (read/rp g p))
  ⊣ foo
    foo-bis
```

Note that RGC context are preserved across different uses of `read/rp`.

`the-context` [bigloo rgc procedure]

Returns the value of the current Rgc context.

10.4 Options and user definitions

Options act as parameters that are transmitted to the parser on the call to `read/rp`. Local defines are user functions inserted in the produced parser, at the same level as the predefined `ignore` function.

Here is an example of grammar using both

```
(define gram
   (regular-grammar (x y)

      (define (foo s)
 (cons* 'foo x s (ignore)))
      (define (bar s)
 (cons* 'bar y s (ignore)))

      ((+ #\a) (foo (the-string)))
      ((+ #\b) (bar (the-string)))
      (else '())))
```

This grammar uses two options *x* and *y*. Hence when invokes it takes two additional values such as:

```
(with-input-from-string "aabb"
   (lambda ()
      (read/rp gram (current-input-port) 'option-x 'option-y)))
   ⇒ (foo option-x aa bar option-y bb)
```

10.5 Examples of regular grammar

The reader who wants to find a real example should read the code of Bigloo's reader. But here are small examples

10.5.1 Word count

The first example presents a grammar that simulates the Unix program `wc`.

```
(let ((*char* 0)
      (*word* 0)
      (*line* 0))
   (regular-grammar ()
      ((+ #\Newline)
       (set! *char* (+ *char* (the-length)))
       (set! *line* (+ *line* (the-length)))
       (ignore))
      ((+ (in #\space #\tab))
       (set! *char* (+ *char* (the-length)))
       (ignore))
      ((+ (out #\newline #\space #\tab))
       (set! *char* (+ *char* (the-length)))
       (set! *word* (+ 1 *word*))
       (ignore))))
```

10.5.2 Roman numbers

The second example presents a grammar that reads Arabic and Roman number.

```
(let ((par-open 0))
   (regular-grammar ((arabic (in ("09")))
                     (roman  (uncase (in "ivxlcdm"))))
      ((+ (in #" \t\n"))
       (ignore))
      ((+ arabic)
       (string->integer (the-string)))
      ((+ roman)
       (roman->arabic (the-string)))
      (#\(
       (let ((open-key par-open))
          (set! par-open (+ 1 par-open))
          (context 'pair)
          (let loop-pair ((walk (ignore)))
             (cond
                ((= open-key par-open)
                 '())
                (else
                 (cons walk (loop-pair (ignore))))))))
      (#\)
       (set! par-open (- par-open 1))
       (if (< par-open 0)
           (begin
              (set! par-open 0)
              (ignore))
           #f))
      ((in "+-*\\")
       (string->symbol (the-string)))
      (else
       (let ((char (the-failure)))
          (if (eof-object? char)
              char
              (error "grammar-roman" "Illegal char" char))))))
```

11 Lalr(1) parsing

Regular grammar generators, like Lex, are often coupled with tools, such as Yacc and Bison, that can generate parsers for more powerful languages, namely (a subset of) context-free languages. These tools take as input a description of the language to be recognized and generate a parser for that language, written in some other language (for example, Yacc and Bison generate parsers written in C). The user must always be aware of the generated parser and that is a nuisance. Bigloo provides such a tool that overcomes this annoyance. It generates parsers for the class of Lalr(1) grammars in a more opaque way.

11.1 Grammar definition

An lalr(1) grammar is defined by the form:

`lalr-grammar` *term-def non-term-def. . .* [bigloo syntax]

term-def is a list of terminal elements of the grammar. Terminals can grouped together to form precedence groups by including the related symbols in a sub-lists of the *term-def* list. Each precedence group must start with one of the keywords `left:`, `right:` or `none:`– this indicates the associativity of the terminal symbol. Here is a sample *term-def* which declares eight terminals:

```
(terminal-1 terminal-2
 (left: terminal-3 terminal-4)
 terminal-5
 (right: terminal-6)
 (none: terminal-7)
 terminal-8)
```

In this case, `terminal-3` and `terminal-4` both have the same precedence, which is greater than the precedence assigned to `terminal-6`. No precedence was assigned to symbols `terminal-1`, `terminal-2`, `terminal-5` or `terminal-8`.

Each *non-term-def* is a list whose first element is the non-terminal being defined, i.e. a symbol. The remaining elements are the production rules associated with this non-terminal. Each rule is a list whose first element is the rule itself (a list of symbols) and the other elements are the semantic actions associated with that particular rule.

For example, consider the following grammar:

```
E ↦ E1 + id {E.val := E1.val + id.val}
      | id {E.val := id.val}
```

With Bigloo, it would be written:

```
(lalr-grammar
  (plus id)
  (e
   ((e plus id)   (+ e id))
   ((id)          id)))
```

The semantic value of a symbol in a rule can be accessed by simply using the name of the symbol in the semantic action associated with the rule. Because a rule can contain multiple occurrences of the same symbol, Bigloo provides a way to access these occurrences separately. To do so, the name of each occurrence must be suffixed by `@`*var* where *var* is the name of a variable that will be bound to the semantic value of the occurrence. For example, if the rule is

```
ifstmt ↦ if E then Stmt else Stmt
```

then, in Bigloo, it would look like

```
(if-stmt
 ((if e then stmt@conseq else stmt@altern)
  (if (eval e)
      (eval conseq)
      (eval altern))))
```

11.2 Precedence and associativity

The bigloo lalr(1) parser generator supports operator precedence and associativity. The method for specifying the precedence for terminal symbols is described in Section 11.1 [Grammar Definition], page 137. Precedence is assigned to each non-terminal production from the precedence of the last terminal symbol appearing in that production.

Typically, when the parser generator encounters a shift/reduce conflict, it produces a warning message, then chooses to reduce. When a parser generator has precedence and associativity information, it can make a much more sophisticated decision.

Let's use this simple calculator grammar as an example:

```
(lalr-grammar
 ((left: op-mult op-div)
  (left: op-add op-sub)
  op-lparen op-rparen
  op-semicolon
  number)

 (file
   (())
   ((file stmt)))
 (stmt
   ((expr op-semicolon) (print expr)))
 (expr
   ((number) number)
   ((expr@a op-add expr@b) (+ a b))
   ((expr@a op-sub expr@b) (- a b))
   ((expr@a op-mult expr@b) (* a b))
   ((expr@a op-div expr@b) (/ a b))
   ((op-lparen expr op-rparen) expr))))
```

Let's start with this input:

```
1 + 2 * 3;
```

At the point where the parser has read `1 + 2` and the lookahead symbol is `*`, the parser encounters a shift/reduce conflict. Should it first reduce by the `(expr op-add expr)` production or shift the `*` in the hopes of reducing the latter expression first?

The `(expr op-add expr)` production has gotten its precedence from the `op-add` terminal symbol. This is the precedence of the reduce. The precedence of the shift comes from the precedence assigned to the lookahead terminal symbol, which is `op-mult`. Since `op-mult` has higher precedence, the parser generator in this state chooses to shift and does not produce a warning.

Here's an example which we can use to demonstrate associativity:

```
1 + 2 - 3;
```

The parser generator encounters a similar shift/reduce conflict this time, except that when it tries to determine whether to shift or reduce, it finds that both actions have the same

precedence. In this case, the parser generator looks at the associativity of the precedence group containing the `op-add` and `op-sub`. Since these are declared to be left-associative, the parser generator chooses to reduce from this state, effectively calculating the `1 + 2`. Had these symbols been right-associative, the parser would have chosen to shift, effectively calculating `2 - 3` first. If these symbols had been declared non-associative with the `none:` keyword, the parser would generate an error if it ever encountered this state.

11.3 The parsing function

Once a grammar has been defined, it can be used to parse some input using the following function:

`read/lalrp` *lg rg port* [*emptyp*] [bigloo procedure]

This function takes three, possibly four, arguments. The first, *lg*, is the Lalr(1) grammar. The second, *rg*, is the lexical analyzer that feeds the grammar with tokens. The third argument, *port*, is the port that contains the input to be parsed. The last argument, *emptyp*, if provided, should be a function of one argument. It is called with each new token read from the port and should return `#t` if the token denotes the end of input. The result of the call is the value computed by the semantic actions of the production rules.

11.4 The regular grammar

In order to work properly, the regular grammar used with an Lalr(1) grammar should follow some conventions:

- If a semantic value is to be associated with the token just parsed, the regular grammar should return a pair whose `car` is the token name (a symbol) and the `cdr` is the semantic value.
- If there is no value associated with the token, the regular grammar can return just the token name. When used in conjunction with an Lalr grammar, regular grammar should never return `#f` as a token value. This is specially true when the regular grammar detects the end of parsing. In that case, the regular grammar *must not* return the `#f` value. A good way to handle end-of-file is illustrated in the following example:

```
(let ((g (regular-grammar ()
             ...
             (else
              (let ((c (the-failure)))
                 (if (eof-object? c)
                     c
                     (error 'rgc "Illegal character" c))))))
      (l (lalr-grammar ...)))
  (read/lalrp l g (current-input-port)))
```

  This way, the Lalr grammar will automatically handles the end-of-file.

11.5 Debugging Lalr Grammars

Currently the debugging facility for debugging Lalr grammars is very limited. When the parameter `bigloo-debug` is set to a value greater or equal to 100, the Lalr engine outputs all of the state changes the parser is going through.

11.6 A simple example

Here is the code for a simple calculator implemented by an Lalr(1) grammar:

```
(begin
  (read/lalrp
   (lalr-grammar
    (nl plus mult minus div const lpar rpar)
    (lines
     (())
     ((lines expression nl)    (display "--> ")
                               (display expression)
                               (newline))
     ((lines nl)))
    (expression
     ((expression plus term)   (+ expression term))
     ((expression minus term)  (- expression term))
     ((term)                   term))
    (term
     ((term mult factor)       (* term factor))
     ((term div factor)        (/ term factor))
     ((factor)                 factor))
    (factor
     ((lpar expression rpar)   expression)
     ((const)                  const)))

   (regular-grammar ()
    ((+ (or #\tab #\space)) (ignore))
    (#\newline              'nl)
    ((+ digit)              (cons 'const (string->number (the-string))))
    (#\+                    'plus)
    (#\-                    'minus)
    (#\*                    'mult)
    (#\/                    'div)
    (#\(                    'lpar)
    (#\)                    'rpar))

   (current-input-port))
  (reset-eof (current-input-port)))
```

12 Posix Regular Expressions

This whole section has been written by **Dorai Sitaram**. It consists in the documentation of the `pregexp` package that may be found at `http://www.ccs.neu.edu/~dorai/pregexp/pregexp.html`.

The regexp notation supported is modeled on Perl's, and includes such powerful directives as numeric and nongreedy quantifiers, capturing and non-capturing clustering, POSIX character classes, selective case- and space-insensitivity, backreferences, alternation, backtrack pruning, positive and negative lookahead and lookbehind, in addition to the more basic directives familiar to all regexp users. A *regexp* is a string that describes a pattern. A regexp matcher tries to *match* this pattern against (a portion of) another string, which we will call the *text string*. The text string is treated as raw text and not as a pattern.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern `"abc"` matches a string that contains the characters `a`, `b`, `c` in succession.

In the regexp pattern, some characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern `"a.c"`, the characters `a` and `c` do stand for themselves but the *metacharacter* `.` can match *any* character (other than newline). Therefore, the pattern `"a.c"` matches an `a`, followed by *any* character, followed by a `c`.

If we needed to match the character `.` itself, we *escape* it, ie, precede it with a backslash (\). The character sequence `\.` is thus a *metasequence*, since it doesn't match itself but rather just `.`. So, to match `a` followed by a literal `.` followed by `c`, we use the regexp pattern `"a\\.c"`.[1] Another example of a metasequence is `\t`, which is a readable way to represent the tab character.

We will call the string representation of a regexp the *U-regexp*, where *U* can be taken to mean *Unix-style* or *universal*, because this notation for regexps is universally familiar. Our implementation uses an intermediate tree-like representation called the *S-regexp*, where *S* can stand for *Scheme*, *symbolic*, or *s-expression*. S-regexps are more verbose and less readable than U-regexps, but they are much easier for Scheme's recursive procedures to navigate.

12.1 Regular Expressions Procedures

Four procedures `pregexp`, `pregexp-match-positions`, `pregexp-match`, `pregexp-replace`, and `pregexp-replace*` enable compilation and matching of regular expressions.

`pregexp` *U-regexp . opt-args* [bigloo procedure]

The procedure `pregexp` takes a U-regexp, which is a string, and returns an S-regexp, which is a tree.

```
(pregexp "c.r") ⇒ (:sub (:or (:seq #\c :any #\r)))
```

[1] The double backslash is an artifact of Scheme strings, not the regexp pattern itself. When we want a literal backslash inside a Scheme string, we must escape it so that it shows up in the string at all. Scheme strings use backslash as the escape character, so we end up with two backslashes — one Scheme-string backslash to escape the regexp backslash, which then escapes the dot. Another character that would need escaping inside a Scheme string is `"`.

There is rarely any need to look at the S-regexps returned by `pregexp`.

The *opt-args* specifies how the regular expression is to be matched. Until documented the argument should be the empty list.

`pregexp-match-positions` *regexp string* [*beg 0*] [*end -1*] [bigloo procedure]

The procedure `pregexp-match-positions` takes a regexp pattern and a text string, and returns a *match* if the pattern *matches* the text string. The pattern may be either a U- or an S-regexp. (`pregexp-match-positions` will internally compile a U-regexp to an S-regexp before proceeding with the matching. If you find yourself calling `pregexp-match-positions` repeatedly with the same U-regexp, it may be advisable to explicitly convert the latter into an S-regexp once beforehand, using `pregexp`, to save needless recompilation.)

`pregexp-match-positions` returns `#f` if the pattern did not match the string; and a list of *index pairs* if it did match. Eg,

```
(pregexp-match-positions "brain" "bird")
 ⇒ #f
(pregexp-match-positions "needle" "hay needle stack")
 ⇒ ((4 . 10))
```

In the second example, the integers 4 and 10 identify the substring that was matched. 1 is the starting (inclusive) index and 2 the ending (exclusive) index of the matching substring.

```
(substring "hay needle stack" 4 10)
 ⇒ "needle"
```

Here, `pregexp-match-positions`'s return list contains only one index pair, and that pair represents the entire substring matched by the regexp. When we discuss *subpatterns* later, we will see how a single match operation can yield a list of *submatches*.

`pregexp-match-positions` takes optional third and fourth arguments that specify the indices of the text string within which the matching should take place.

```
(pregexp-match-positions "needle"
  "his hay needle stack -- my hay needle stack -- her hay needle stack"
  24 43)
 ⇒ ((31 . 37))
```

Note that the returned indices are still reckoned relative to the full text string.

`pregexp-match` *regexp string* [bigloo procedure]

The procedure `pregexp-match` is called like `pregexp-match-positions` but instead of returning index pairs it returns the matching substrings:

```
(pregexp-match "brain" "bird")
 ⇒ #f
(pregexp-match "needle" "hay needle stack")
 ⇒ ("needle")
```

`pregexp-match` also takes optional third and fourth arguments, with the same meaning as does `pregexp-match-positions`.

`pregexp-replace` *regexp string1 string2* [bigloo procedure]

The procedure `pregexp-replace` replaces the matched portion of the text string by another string. The first argument is the regexp, the second the text string, and the third is the *insert string* (string to be inserted).

```
(pregexp-replace "te" "liberte" "ty")
```

```
⇒ "liberty"
```

If the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to the text string.

`pregexp-replace*` *regexp string1 string2* [bigloo procedure]

The procedure `pregexp-replace*` replaces *all* matches in the text *string1* by the insert *string2*:

```
(pregexp-replace* "te" "liberte egalite fraternite" "ty")
 ⇒ "liberty egality fratyrnity"
```

As with `pregexp-replace`, if the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to the text string.

`pregexp-split` *regexp string* [bigloo procedure]

The procedure `pregexp-split` takes two arguments, a regexp pattern and a text string, and returns a list of substrings of the text string, where the pattern identifies the delimiter separating the substrings.

```
(pregexp-split ":" "/bin:/usr/bin:/usr/bin/X11:/usr/local/bin")
 ⇒ ("/bin" "/usr/bin" "/usr/bin/X11" "/usr/local/bin")

(pregexp-split " " "pea soup")
 ⇒ ("pea" "soup")
```

If the first argument can match an empty string, then the list of all the single-character substrings is returned.

```
(pregexp-split "" "smithereens")
 ⇒ ("s" "m" "i" "t" "h" "e" "r" "e" "e" "n" "s")
```

To identify one-or-more spaces as the delimiter, take care to use the regexp `" +"`, not `" *"`.

```
(pregexp-split " +" "split pea     soup")
 ⇒ ("split" "pea" "soup")

(pregexp-split " *" "split pea     soup")
 ⇒ ("s" "p" "l" "i" "t" "p" "e" "a" "s" "o" "u" "p")
```

`pregexp-quote` *string* [bigloo procedure]

The procedure `pregexp-quote` takes an arbitrary *string* and returns a U-regexp (string) that precisely represents it. In particular, characters in the input string that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

```
(pregexp-quote "cons")
 ⇒ "cons"

(pregexp-quote "list?")
 ⇒ "list\\?"
```

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

12.2 Regular Expressions Pattern Language

Here is a complete description of the regexp pattern language recognized by the `pregexp` procedures.

12.2.1 Basic assertions

The *assertions* `^` and `$` identify the beginning and the end of the text string respectively. They ensure that their adjoining regexps match at one or other end of the text string. Examples:

```
(pregexp-match-positions "^contact" "first contact") ⇒ #f
```

The regexp fails to match because `contact` does not occur at the beginning of the text string.

```
(pregexp-match-positions "laugh$" "laugh laugh laugh laugh") ⇒ ((18 . 23))
```

The regexp matches the *last* `laugh`.

The metasequence `\b` asserts that a *word boundary* exists.

```
(pregexp-match-positions "yack\\b" "yackety yack") ⇒ ((8 . 12))
```

The `yack` in `yackety` doesn't end at a word boundary so it isn't matched. The second `yack` does and is.

The metasequence `\B` has the opposite effect to `\b`. It asserts that a word boundary does not exist.

```
(pregexp-match-positions "an\\B" "an analysis") ⇒ ((3 . 5))
```

The `an` that doesn't end in a word boundary is matched.

12.2.2 Characters and character classes

Typically a character in the regexp matches the same character in the text string. Sometimes it is necessary or convenient to use a regexp metasequence to refer to a single character. Thus, metasequences `\n`, `\r`, `\t`, and `\.` match the newline, return, tab and period characters respectively.

The *metacharacter* period (`.`) matches *any* character other than newline.

```
(pregexp-match "p.t" "pet") ⇒ ("pet")
```

It also matches `pat`, `pit`, `pot`, `put`, and `p8t` but not `peat` or `pfffft`.

A *character class* matches any one character from a set of characters. A typical format for this is the *bracketed character class* `[...]`, which matches any one character from the non-empty sequence of characters enclosed within the brackets.[2] Thus `"p[aeiou]t"` matches `pat`, `pet`, `pit`, `pot`, `put` and nothing else.

Inside the brackets, a hyphen (`-`) between two characters specifies the ascii range between the characters. Eg, `"ta[b-dgn-p]"` matches `tab`, `tac`, `tad`, *and* `tag`, *and* `tan`, `tao`, `tap`.

An initial caret (`^`) after the left bracket inverts the set specified by the rest of the contents, ie, it specifies the set of characters *other than* those identified in the brackets. Eg, `"do[^g]"` matches all three-character sequences starting with `do` except `dog`.

Note that the metacharacter `^` inside brackets means something quite different from what it means outside. Most other metacharacters (`.`, `*`, `+`, `?`, etc) cease to be metacharacters when inside brackets, although you may still escape them for peace of mind. `-` is a metacharacter only when it's inside brackets, and neither the first nor the last character.

Bracketed character classes cannot contain other bracketed character classes (although they contain certain other types of character classes — see below). Thus a left bracket (`[`)

[2] Requiring a bracketed character class to be non-empty is not a limitation, since an empty character class can be more easily represented by an empty string.

inside a bracketed character class doesn't have to be a metacharacter; it can stand for itself. Eg, `"[a[b]"` matches `a`, `[`, and `b`.

Furthermore, since empty bracketed character classes are disallowed, a right bracket (`]`) immediately occurring after the opening left bracket also doesn't need to be a metacharacter. Eg, `"[]ab]"` matches `]`, `a`, and `b`.

12.2.3 Some frequently used character classes

Some standard character classes can be conveniently represented as metasequences instead of as explicit bracketed expressions. `\d` matches a digit (`[0-9]`); `\s` matches a whitespace character; and `\w` matches a character that could be part of a "word".[3]

The upper-case versions of these metasequences stand for the inversions of the corresponding character classes. Thus `\D` matches a non-digit, `\S` a non-whitespace character, and `\W` a non-"word" character.

Remember to include a double backslash when putting these metasequences in a Scheme string:

```
(pregexp-match "\\d\\d" "0 dear, 1 have 2 read catch 22 before 9") ⇒ ("22")
```

These character classes can be used inside a bracketed expression. Eg, `"[a-z\\d]"` matches a lower-case letter or a digit.

12.2.4 POSIX character classes

A *POSIX character class* is a special metasequence of the form `[:...:]` that can be used only inside a bracketed expression. The POSIX classes supported are

```
[:alnum:]  letters and digits
[:alpha:]  letters
[:algor:]  the letters c, h, a and d
[:ascii:]  7-bit ascii characters
[:blank:]  widthful whitespace, ie, space and tab
[:cntrl:]  ‘‘control’’ characters, viz, those with code < 32
[:digit:]  digits, same as \d
[:graph:]  characters that use ink
[:lower:]  lower-case letters
[:print:]  ink-users plus widthful whitespace
[:space:]  whitespace, same as \s
[:upper:]  upper-case letters
[:word:]   letters, digits, and underscore, same as \w
[:xdigit:] hex digits
```

For example, the regexp `"[[:alpha:]_]"` matches a letter or underscore.

```
(pregexp-match "[[:alpha:]_]" "--x--") ⇒ ("x")
(pregexp-match "[[:alpha:]_]" "--_--") ⇒ ("_")
(pregexp-match "[[:alpha:]_]" "--:--") ⇒ #f
```

The POSIX class notation is valid *only* inside a bracketed expression. For instance, `[:alpha:]`, when not inside a bracketed expression, will *not* be read as the letter class. Rather it is (from previous principles) the character class containing the characters `:`, `a`, `l`, `p`, `h`.

```
(pregexp-match "[[:alpha:]]" "--a--") ⇒ ("a")
(pregexp-match "[[:alpha:]]" "--_--") ⇒ #f
```

[3] Following regexp custom, we identify "word" characters as `[A-Za-z0-9_]`, although these are too restrictive for what a Schemer might consider a "word".

By placing a caret (`^`) immediately after `[:`, you get the inversion of that POSIX character class. Thus, `[:^alpha]` is the class containing all characters except the letters.

12.2.5 Quantifiers

The *quantifiers* `*`, `+`, and `?` match respectively: zero or more, one or more, and zero or one instances of the preceding subpattern.

```
(pregexp-match-positions "c[ad]*r" "cadaddadddr") ⇒ ((0 . 11))
(pregexp-match-positions "c[ad]*r" "cr")          ⇒ ((0 . 2))

(pregexp-match-positions "c[ad]+r" "cadaddadddr") ⇒ ((0 . 11))
(pregexp-match-positions "c[ad]+r" "cr")          ⇒ #f

(pregexp-match-positions "c[ad]?r" "cadaddadddr") ⇒ #f
(pregexp-match-positions "c[ad]?r" "cr")          ⇒ ((0 . 2))
(pregexp-match-positions "c[ad]?r" "car")         ⇒ ((0 . 3))
```

12.2.6 Numeric quantifiers

You can use braces to specify much finer-tuned quantification than is possible with `*`, `+`, `?`.

The quantifier `{m}` matches *exactly* `m` instances of the preceding *subpattern*. `m` must be a nonnegative integer.

The quantifier `{m,n}` matches at least `m` and at most `n` instances. `m` and `n` are nonnegative integers with `m <= n`. You may omit either or both numbers, in which case `m` defaults to 0 and `n` to infinity.

It is evident that `+` and `?` are abbreviations for `{1,}` and `{0,1}` respectively. `*` abbreviates `{,}`, which is the same as `{0,}`.

```
(pregexp-match "[aeiou]{3}" "vacuous")  ⇒ ("uou")
(pregexp-match "[aeiou]{3}" "evolve")   ⇒ #f
(pregexp-match "[aeiou]{2,3}" "evolve") ⇒ #f
(pregexp-match "[aeiou]{2,3}" "zeugma") ⇒ ("eu")
```

12.2.7 Non-greedy quantifiers

The quantifiers described above are *greedy*, ie, they match the maximal number of instances that would still lead to an overall match for the full pattern.

```
(pregexp-match "<.*>" "<tag1> <tag2> <tag3>")
 ⇒ ("<tag1> <tag2> <tag3>")
```

To make these quantifiers *non-greedy*, append a `?` to them. Non-greedy quantifiers match the minimal number of instances needed to ensure an overall match.

```
(pregexp-match "<.*?>" "<tag1> <tag2> <tag3>") ⇒ ("<tag1>")
```

The non-greedy quantifiers are respectively: `*?`, `+?`, `??`, `{m}?`, `{m,n}?`. Note the two uses of the metacharacter `?`.

12.2.8 Clusters

Clustering, ie, enclosure within parens `(...)`, identifies the enclosed *subpattern* as a single entity. It causes the matcher to *capture* the *submatch*, or the portion of the string matching the subpattern, in addition to the overall match.

```
(pregexp-match "([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
 ⇒ ("jan 1, 1970" "jan" "1" "1970")
```

Clustering also causes a following quantifier to treat the entire enclosed subpattern as an entity.

```
(pregexp-match "(poo )*" "poo poo platter") ⇒ ("poo poo " "poo ")
```

The number of submatches returned is always equal to the number of subpatterns specified in the regexp, even if a particular subpattern happens to match more than one substring or no substring at all.

```
(pregexp-match "([a-z ]+;)*" "lather; rinse; repeat;")
 ⇒ ("lather; rinse; repeat;" " repeat;")
```

Here the `*`-quantified subpattern matches three times, but it is the last submatch that is returned.

It is also possible for a quantified subpattern to fail to match, even if the overall pattern matches. In such cases, the failing submatch is represented by `#f`.

```
(define date-re
  ;match 'month year' or 'month day, year'.
  ;subpattern matches day, if present
  (pregexp "([a-z]+) +([0-9]+,)? *([0-9]+)"))

(pregexp-match date-re "jan 1, 1970")
 ⇒ ("jan 1, 1970" "jan" "1," "1970")

(pregexp-match date-re "jan 1970")
 ⇒ ("jan 1970" "jan" #f "1970")
```

12.2.9 Backreferences

Submatches can be used in the insert string argument of the procedures `pregexp-replace` and `pregexp-replace*`. The insert string can use `\n` as a *backreference* to refer back to the nth submatch, ie, the substring that matched the nth subpattern. `\0` refers to the entire match, and it can also be specified as `\&`.

```
(pregexp-replace "_(.+?)_"
  "the _nina_, the _pinta_, and the _santa maria_"
  "*\\1*")
 ⇒ "the *nina*, the _pinta_, and the _santa maria_"

(pregexp-replace* "_(.+?)_"
  "the _nina_, the _pinta_, and the _santa maria_"
  "*\\1*")
 ⇒ "the *nina*, the *pinta*, and the *santa maria*"

;recall: \S stands for non-whitespace character

(pregexp-replace "(\\S+) (\\S+) (\\S+)"
  "eat to live"
  "\\3 \\2 \\1")
 ⇒ "live to eat"
```

Use `\\` in the insert string to specify a literal backslash. Also, `\$` stands for an empty string, and is useful for separating a backreference `\n` from an immediately following number.

Backreferences can also be used within the regexp pattern to refer back to an already matched subpattern in the pattern. `\n` stands for an exact repeat of the nth submatch.[4]

```
(pregexp-match "([a-z]+) and \\1"
```

[4] `\0`, which is useful in an insert string, makes no sense within the regexp pattern, because the entire regexp has not matched yet that you could refer back to it.

```
  "billions and billions")
 ⇒ ("billions and billions" "billions")
```

Note that the backreference is not simply a repeat of the previous subpattern. Rather it is a repeat of *the particular substring already matched by the subpattern*.

In the above example, the backreference can only match `billions`. It will not match `millions`, even though the subpattern it harks back to — `([a-z]+)` — would have had no problem doing so:

```
(pregexp-match "([a-z]+) and \\1"
  "billions and millions")
 ⇒ #f
```

The following corrects doubled words:

```
(pregexp-replace* "(\\S+) \\1"
  "now is the the time for all good men to to come to the aid of of the party"
  "\\1")
 ⇒ "now is the time for all good men to come to the aid of the party"
```

The following marks all immediately repeating patterns in a number string:

```
(pregexp-replace* "(\\d+)\\1"
  "12334098324243242098098234"
  "{\\1,\\1}")
 ⇒ "12{3,3}40983{24,24}3242{098,098}0234"
```

12.2.10 Non-capturing clusters

It is often required to specify a cluster (typically for quantification) but without triggering the capture of submatch information. Such clusters are called *non-capturing*. In such cases, use `(?:` instead of `(` as the cluster opener. In the following example, the non-capturing cluster eliminates the "directory" portion of a given pathname, and the capturing cluster identifies the basename.

```
(pregexp-match "^(?:[a-z]*/)*([a-z]+)$"
  "/usr/local/bin/mzscheme")
 ⇒ ("/usr/local/bin/mzscheme" "mzscheme")
```

12.2.11 Cloisters

The location between the `?` and the `:` of a non-capturing cluster is called a *cloister*.[5] You can put *modifiers* there that will cause the enclustered subpattern to be treated specially. The modifier `i` causes the subpattern to match *case-insensitively*:

```
(pregexp-match "(?i:hearth)" "HeartH") ⇒ ("HeartH")
```

The modifier `x` causes the subpattern to match *space-insensitively*, ie, spaces and comments within the subpattern are ignored. Comments are introduced as usual with a semicolon (`;`) and extend till the end of the line. If you need to include a literal space or semicolon in a space-insensitized subpattern, escape it with a backslash.

```
(pregexp-match "(?x: a   lot)" "alot")
 ⇒ ("alot")

(pregexp-match "(?x: a  \\  lot)" "a lot")
 ⇒ ("a lot")

(pregexp-match "(?x:
```

[5] A useful, if terminally cute, coinage from the abbots of Perl.

```
  a \\ man  \\; \\   # ignore
  a \\ plan \\; \\   # me
  a \\ canal         # completely
  )"
 "a man; a plan; a canal")
 ⇒ ("a man; a plan; a canal")
```

You can put more than one modifier in the cloister.

```
(pregexp-match "(?ix:
  a \\ man  \\; \\   # ignore
  a \\ plan \\; \\   # me
  a \\ canal         # completely
  )"
 "A Man; a Plan; a Canal")
 ⇒ ("A Man; a Plan; a Canal")
```

A minus sign before a modifier inverts its meaning. Thus, you can use `-i` and `-x` in a *subcluster* to overturn the insensitivities caused by an enclosing cluster.

```
(pregexp-match "(?i:the (?-i:TeX)book)"
  "The TeXbook")
 ⇒ ("The TeXbook")
```

This regexp will allow any casing for `the` and `book` but insists that `TeX` not be differently cased.

12.2.12 Alternation

You can specify a list of *alternate* subpatterns by separating them by `|`. The `|` separates subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no enclosing parens).

```
(pregexp-match "f(ee|i|o|um)" "a small, final fee")
 ⇒ ("fi" "i")

(pregexp-replace* "([yi])s(e[sdr]?|ing|ation)"
   "it is energising to analyse an organisation
   pulsing with noisy organisms"
   "\\1z\\2")
 ⇒ "it is energizing to analyze an organization
   pulsing with noisy organisms"
```

Note again that if you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use `(?:` instead of `(`.

```
(pregexp-match "f(?:ee|i|o|um)" "fun for all")
 ⇒ ("fo")
```

An important thing to note about alternation is that the leftmost matching alternate is picked regardless of its length. Thus, if one of the alternates is a prefix of a later alternate, the latter may not have a chance to match.

```
(pregexp-match "call|call-with-current-continuation"
  "call-with-current-continuation")
 ⇒ ("call")
```

To allow the longer alternate to have a shot at matching, place it before the shorter one:

```
(pregexp-match "call-with-current-continuation|call"
  "call-with-current-continuation")
 ⇒ ("call-with-current-continuation")
```

In any case, an overall match for the entire regexp is always preferred to an overall nonmatch. In the following, the longer alternate still wins, because its preferred shorter prefix fails to yield an overall match.

```
(pregexp-match "(?:call|call-with-current-continuation) constrained"
  "call-with-current-continuation constrained")
 ⇒ ("call-with-current-continuation constrained")
```

12.2.13 Backtracking

We've already seen that greedy quantifiers match the maximal number of times, but the overriding priority is that the overall match succeed. Consider

```
(pregexp-match "a*a" "aaaa")
```

The regexp consists of two subregexps, `a*` followed by `a`. The subregexp `a*` cannot be allowed to match all four `a`'s in the text string `"aaaa"`, even though `*` is a greedy quantifier. It may match only the first three, leaving the last one for the second subregexp. This ensures that the full regexp matches successfully.

The regexp matcher accomplishes this via a process called *backtracking*. The matcher tentatively allows the greedy quantifier to match all four `a`'s, but then when it becomes clear that the overall match is in jeopardy, it *backtracks* to a less greedy match of *three* `a`'s. If even this fails, as in the call

```
(pregexp-match "a*aa" "aaaa")
```

the matcher backtracks even further. Overall failure is conceded only when all possible backtracking has been tried with no success.

Backtracking is not restricted to greedy quantifiers. Nongreedy quantifiers match as few instances as possible, and progressively backtrack to more and more instances in order to attain an overall match. There is backtracking in alternation too, as the more rightward alternates are tried when locally successful leftward ones fail to yield an overall match.

12.2.14 Disabling backtracking

Sometimes it is efficient to disable backtracking. For example, we may wish to *commit* to a choice, or we know that trying alternatives is fruitless. A nonbacktracking regexp is enclosed in `(?>...)`.

```
(pregexp-match "(?>a+)." "aaaa")
 ⇒ #f
```

In this call, the subregexp `?>a*` greedily matches all four `a`'s, and is denied the opportunity to backpedal. So the overall match is denied. The effect of the regexp is therefore to match one or more `a`'s followed by something that is definitely non-`a`.

12.2.15 Looking ahead and behind

You can have assertions in your pattern that look *ahead* or *behind* to ensure that a subpattern does or does not occur. These "look around" assertions are specified by putting the subpattern checked for in a cluster whose leading characters are: `?=` (for positive lookahead), `?!` (negative lookahead), `?<=` (positive lookbehind), `?<!` (negative lookbehind). Note that the subpattern in the assertion does not generate a match in the final result. It merely allows or disallows the rest of the match.

12.2.16 Lookahead

Positive lookahead (`?=`) peeks ahead to ensure that its subpattern *could* match.

```
(pregexp-match-positions "grey(?=hound)"
  "i left my grey socks at the greyhound")
 ⇒ ((28 . 32))
```

The regexp `"grey(?=hound)"` matches `grey`, but *only* if it is followed by `hound`. Thus, the first `grey` in the text string is not matched.

Negative lookahead (`?!`) peeks ahead to ensure that its subpattern could not possibly match.

```
(pregexp-match-positions "grey(?!hound)"
  "the gray greyhound ate the grey socks")
 ⇒ ((27 . 31))
```

The regexp `"grey(?!hound)"` matches `grey`, but only if it is *not* followed by `hound`. Thus the `grey` just before `socks` is matched.

12.2.17 Lookbehind

Positive lookbehind (`?<=`) checks that its subpattern *could* match immediately to the left of the current position in the text string.

```
(pregexp-match-positions "(?<=grey)hound"
  "the hound in the picture is not a greyhound")
 ⇒ ((38 . 43))
```

The regexp `(?<=grey)hound` matches `hound`, but only if it is preceded by `grey`.

Negative lookbehind (`?<!`) checks that its subpattern could not possibly match immediately to the left.

```
(pregexp-match-positions "(?<!grey)hound"
  "the greyhound in the picture is not a hound")
 ⇒ ((38 . 43))
```

The regexp `(?<!grey)hound` matches `hound`, but only if it is *not* preceded by `grey`.

Lookaheads and lookbehinds can be convenient when they are not confusing.

12.3 An Extended Example

Here's an extended example from Friedl that covers many of the features described above. The problem is to fashion a regexp that will match any and only IP addresses or *dotted quads*, ie, four numbers separated by three dots, with each number between 0 and 255. We will use the commenting mechanism to build the final regexp with clarity. First, a subregexp `n0-255` that matches 0 through 255.

```
(define n0-255
  "(?x:
  \\d          ;  0 through   9
  | \\d\\d     ; 00 through  99
  | [01]\\d\\d ;000 through 199
  | 2[0-4]\\d  ;200 through 249
  | 25[0-5]    ;250 through 255
  )")
```

The first two alternates simply get all single- and double-digit numbers. Since 0-padding is allowed, we need to match both 1 and 01. We need to be careful when getting 3-digit

numbers, since numbers above 255 must be excluded. So we fashion alternates to get 000 through 199, then 200 through 249, and finally 250 through 255.[6]

An IP-address is a string that consists of four `n0-255`s with three dots separating them.

```
(define ip-re1
  (string-append
    "^"         ;nothing before
    n0-255      ;the first n0-255,
    "(?x:"      ;then the subpattern of
    "\\."       ;a dot followed by
    n0-255      ;an n0-255,
    ")"         ;which is
    "{3}"       ;repeated exactly 3 times
    "$"         ;with nothing following
    ))
```

Let's try it out.

```
(pregexp-match ip-re1 "1.2.3.4")        ⇒ ("1.2.3.4")
(pregexp-match ip-re1 "55.155.255.265") ⇒ #f
```

which is fine, except that we also have

```
(pregexp-match ip-re1 "0.00.000.00") ⇒ ("0.00.000.00")
```

All-zero sequences are not valid IP addresses! Lookahead to the rescue. Before starting to match `ip-re1`, we look ahead to ensure we don't have all zeros. We could use positive lookahead to ensure there *is* a digit other than zero.

```
(define ip-re
  (string-append
    "(?=.*[1-9])" ;ensure there's a non-0 digit
    ip-re1))
```

Or we could use negative lookahead to ensure that what's ahead isn't composed of *only* zeros and dots.

```
(define ip-re
  (string-append
    "(?![0.]*$)" ;not just zeros and dots
                 ;(note: dot is not metachar inside [])
    ip-re1))
```

The regexp `ip-re` will match all and only valid IP addresses.

```
(pregexp-match ip-re "1.2.3.4") ⇒ ("1.2.3.4")
(pregexp-match ip-re "0.0.0.0") ⇒ #f
```

[6] Note that `n0-255` lists prefixes as preferred alternates, something we cautioned against in section Section 12.2.12 [Alternation], page 149. However, since we intend to anchor this subregexp explicitly to force an overall match, the order of the alternates does not matter.

13 Command Line Parsing

Bigloo supports command line argument parsing. That is, when an application is spawn from an Unix shell, the `main` function is called and its argument is bound to the list of the command line arguments, See Chapter 2 [Module declaration], page 7. The `args-parse` form may be used to parse these.

`args-parse` *list rules* [*null-rule*] [*else-rule*] . . . [bigloo syntax]

The argument *list* is a list of strings. *Rules* is defined by the following grammar:

```
<rule>      ↦ (section <string>)
              | ((<option> <help>) <s-expression>)
              | ((<option>) <s-expression>)
              | ((<flag> <var> <var> ...) <s-expression>)
              | ((<flag> <var> <var> ... <help>) <s-expression>)
<null-rule> ↦ (() <s-expression>)
<else-rule> ↦ (else <s-expression>)
<option>    ↦ <flag>
              | <string><var>
<flag>      ↦ <string>
              | (<string>+)
<var>       ↦ an identifier leaded by the ? character
<help>      ↦ (help <s-expression>)
              | (help <string> <s-expression>)
```

Each elements of *list* are match against the *rules*. If one of these matches, `args-parse` proceeds as follows:

1. The matched argument elements of *list* are removed from the list.
2. The `<s-expression>` associated to the matching rule is evaluated in an environment where the rule variables are bound.
3. The argument parsing is resumed with the rest of *list*.

In addition to parsing the command line arguments, `args-parse` enables help message printing.

`args-parse-usage` *fmt* [bigloo procedure]

This is a procedure of one argument, an boolean. `Args-parse-usage` constructs an help message from all the option described in a `args-parse` form. `Args-parse-usage` is only defined in the `<s-expression>` of an `args-parse` form.

At last, if no rule matches an argument and if the `args-parse` form contains an `else` rule, this is evaluated. In the `<s-expression>` part of that rule, the pseudo-variable `else` is bound to the first unmatched argument and the pseudo-variable `rest` is bound to all the unmatched arguments.

Here is an example of argument parsing deploying all the possible rules:

```
(module args-example
   (main main))

(define (main argv)
   (args-parse (cdr argv)
      (section "Help")
      (("?")
       (args-parse-usage #f))
```

```
((("-h" "--help") (help "?,-h,--help" "This help message"))
 (args-parse-usage #f))
(section "Misc")
((("-v" "--version") (help "Version number"))
 (print *version*))
(("-o" ?file (help "The output file"))
 (set! *dest* file))
(("--input=?file" (help "The input file"))
 (set! *input* file))
(else
 (print "Illegal argument `" else "'. Usage:")
 (args-parse-usage #f))))
```

Invoking the compiled `args-example` module could produce:

```
> bigloo.new args.scm
args.scm:
> a.out toto
Illegal argument 'toto'. Usage:

Help:
  ?,-h,--help    --  This help message

Misc:
  -v,--version   --  Version number
  -o <file>      --  The output file
  --input=<file> --  The input file
```

14 Cryptography

Bigloo provides several functions for encrypting and decrypting documents. These are described in the chapter. Unless explicitly mentioned all functions presented in this document are accessible via the `crypto` library.

None of the cryptographic functions are protected against timing attacks. No effort has been spent on protecting used memory.

Here is an example of a module that uses this library:

```
;; Encrypt a string using AES.
(module aes-encrypt
   (library crypto)
   (main main))

(define (main argv)
   (when (and (pair? (cdr argv)) (pair? (cddr argv)))
      (let ((encrypt? (string=? "-e" (cadr argv)))
            (passwd (caddr argv))
            (input (read-string)))
         (if encrypt?
             (display (encrypt 'aes input passwd))
             (display (decrypt 'aes input passwd))))))
```

14.1 Symmetric Block Ciphers

Bigloo supports some common block ciphers. Block ciphers work on blocks of fixed size. A *mode of operation* defines the way bigger input is handled. For instance in ECB (Electronic Codebook mode) the blocks are all encrypted separately, whereas CBC (Cipher-Block Chaining) chains all blocks.

All modes that chain the blocks need an IV (Initial Vector) to "bootstrap" the chaining.

Block ciphers by themselves can only work on full blocks. Some modes are constructed in a way that even incomplete blocks can be safely processed. For the remaining blocks a padding function needs to be given.

Most block ciphers only work with keys of specific length. The following functions take passwords (strings of arbitrary length) as input, and preprocess the given password by a *:string->key* function. The result must then be of correct length.

`encrypt::bstring` *cipher plain password* [Bigloo Cryptography procedure]
[*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

`encrypt-string::bstring` *cipher* [Bigloo Cryptography procedure]
plaintext::bstring password [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

`encrypt-mmap::bstring` *cipher* [Bigloo Cryptography procedure]
plaintext::mmap password [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

`encrypt-port::bstring` *cipher* [Bigloo Cryptography procedure]
plaintext::input-port password [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

`encrypt-file::bstring` *cipher* [Bigloo Cryptography procedure]
filename::bstring password [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

`encrypt-sendchars` *cipher in::input-port* [Bigloo Cryptography procedure]
out::output-port password [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

The procedure `encrypt` encrypts its input using the chosen *cipher*. The result is returned as string. `encrypt` dispatches depending on the type of *plain*. Strings are processed by `encrypt-string` (and not `encrypt-file`).

The function `encrypt-sendchars` reads from an input-port *in* and encrypts its output directly into an output-port *out*.

The symbol *cipher* can be one of:

- `des`: Data Encryption Standard (DES). DES works on blocks of 64 bits. DES requires keys of length 64 (bits), but only 56 of these bits are actually used. Bigloo's implementation therefore accepts both. *DES is considered to be insecure and its usage is discouraged.*
- `des3`: Triple DES, Triple Data Encryption Algorithm (DES3, TDEA). DES3 works on blocks of 64 bits. DES3 requires keys of length 128 or 192 (bits), but only 112/168 of these bits are actually used. Bigloo's implementation therefore accepts the smaller keys too.

  Bigloo's DES3 implementation has been changed with release 3.4b. Earlier versions did not use the full key for en/decryption.
- `des-np`: Same as `des`, but the initial and final permutations are not performed.
- `des3-np`: Same as `des3`, but the initial and final permutations are not performed.
- `aes`: Advanced Encryption Standard (AES). AES works on blocks of 128 bits. AES requires keys of length 128, 192 or 256 bits.
- `cast-128`: CAST-128 (CAST5). CAST-128 works on blocks of 64 bits. CAST-128 requires a key-length of 40-128 bits.
- `idea`: International Data Encryption Algorithm (IDEA). IDEA works on blocks of 64 bits. It requires keys of length 128 (in bits). *IDEA is patented in many countries (including the USA and most European countries) but it is free for non-commercial use.*

The given password must be a string. An optional parameter *:string->key* should transform this password so that it has the correct length for the cipher. A small list of possible functions are provided in the Section 14.1.1 [String to Key], page 159, section.

By default `string->key-hash` with SHA-1 will be used. The key-length will depend on the chosen cipher:

- `des`: 56 bits.
- `des3`: 112 bits.
- `des-np`: Same as `des`.
- `des3-np`: Same as `des3`.
- `aes`: 192 bits.

- `cast-128`: 128 bits.
- `idea`: 128 bits.

Bigloo supports the following block cipher modes (*:mode*):

- `ecb`: Electronic codebook.
- `cbc`: Cipher-block chaining.
- `pcbc`: Propagating cipher-block chaining.
- `cfb`: Cipher feedback.
- `ofb`: Output feedback.
- `ctr`: Counter.

By default `cfb` is chosen.

Electronic codebook mode en/decodes each block independently and is hence the closest to the block cipher. It is however inherently unsafe as blocks with the same content are encrypted to the same output.

With the exception of `ecb` all other modes can be initialized with an IV (Initialization vector). If *:IV* is false, then a random one will be generated. During encryption this randomly generated IV will be prefixed to the result. When calling the decryption routine without any IV the procedure will use the first block of the input as IV.

In `ctr` (counter) mode the IV parameter serves as nonce. Two additional key-parameters `:nonce-init` and `:nonce-update` are then used to initialize and update the block-sized nonce string. Before encrypting the first block `nonce-init` will be invoked with an empty block-sized string and the initial nonce (IV). It must initialize the string with the nonce. For each block `nonce-update` will be called with the string, the nonce, and the number of already encrypted blocks (hence 0 at the very beginning). By default `nonce-init` takes the IV-*string* and blits it into the given string. `nonce-update` simply increments the string (treating the given string as one big number).

Note that the initial nonce (passed using IV) may be of any type. As long as `nonce-init` and `nonce-update` correctly initialize and update the passed string.

The input's length of modes `ecb`, `cbc` and `pcbc` must be a multiple of the block-size. Should this not be the case a padding algorithm must be specified (`:pad`). Currently are implemented (examples for hexadecimal string "DD" and cipher block size 4):

- `none`: No padding. Raises an error should the input not be a multiple.
- `bit`: Bit padding. Add a '1' bit and then '0' bits. Example: "DD 80 00 00".
- `ansi-x.923`: Byte padding. Fill with #x00s followed by the number of added bytes (the counter inclusive). Example: "DD 00 00 03".
- `iso-10126`: Fill with random characters followed by the number of added bytes (the counter inclusive). Example: "DD 42 31 03".
- `pkcs7`: Fill with the number of added bytes. Example: "DD 03 03 03".
- `zero`: Fill with zeros. This is only reversible if the input is guaranteed not to finish with a zero character. Example: "DD 00 00 00".

Alternatively users can supply their own (un)pad functions (instead of a symbol). The signature of a padding function is `(pad::bool str::bstring valid-chars::long)`.

It receives the last block of the input. Should the input be of correct length then the an empty block will be sent to the padding function. `valid-chars` indicates the number of read characters. It ranges from 0 to blocksize-1. The padding function should fill the block and return `#t` if this last block should be encoded. By returning `#f` the last block will be discarded. This makes only sense if `valid-chars` was equal to 0.

The unpadding procedure has the signature `(unpad::long str::bstring)`. The input string will have the length of the block-size. The unpadding function may modify the string and must return the number of characters that are valid.

`decrypt::bstring` *cipher ciphertext password* [Bigloo Cryptography procedure]
[*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*]

`decrypt-string::bstring` *cipher ciphertext::bstring password* [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*] [Bigloo Cryptography procedure]

`decrypt-mmap::bstring` *cipher ciphertext::mmap password* [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*] [Bigloo Cryptography procedure]

`decrypt-port::bstring` *cipher ciphertext::input-port password* [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*] [Bigloo Cryptography procedure]

`decrypt-file::bstring` *cipher filename::bstring password* [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*] [Bigloo Cryptography procedure]

`decrypt-sendchars` *cipher in::input-port out::output-port password* [*:string->key*] [*:mode 'cfb*] [*:IV #f*] [*:pad 'none*] [*:nonce-init!*] [*:nonce-update!*] [Bigloo Cryptography procedure]

Counterpart to the encryption functions. With the same parameters the `decrypt` function will decrypt the result of an `encrypt` call. Without *:IV* (Initial Vector) the `decrypt` function will use the first block as IV.

For compatibility the following functions remain in Bigloo. They are in the default library and not inside the `crypto` library.

`aes-ctr-encrypt` *text password* [*nbits 128*] [bigloo procedure]
`aes-ctr-encrypt-mmap` *mmap password* [*nbits 128*] [bigloo procedure]
`aes-ctr-encrypt-string` *string password* [*nbits 128*] [bigloo procedure]
`aes-ctr-encrypt-port` *iport password* [*nbits 128*] [bigloo procedure]
`aes-ctr-encrypt-file` *filename password* [*nbits 128*] [bigloo procedure]

These functions are equivalent to a call to `aes-encrypt` with mode set to `ctr` and a special `:string->key` parameter. The optional argument *nbits* must either be `128`, `192`, or `256` and determines the size of the key.

`aes-ctr-decrypt` *text password* [*nbits 128*] [bigloo procedure]
`aes-ctr-decrypt-mmap` *mmap password* [*nbits 128*] [bigloo procedure]
`aes-ctr-decrypt-string` *string password* [*nbits 128*] [bigloo procedure]
`aes-ctr-decrypt-port` *iport password* [*nbits 128*] [bigloo procedure]

`aes-ctr-decrypt-file` *filename password* [*nbits 128*] [bigloo procedure]
Counterpart to `aes-ctr-encrypt`.

14.1.1 String to Key

The following string->key algorithms take a password string and transform it to a key string of a given length. In all the functions the *len* is expressed in bytes.

`string->key-zero` *str len* [Bigloo Cryptography procedure]
If the length of the input string *str* is greater or equal to *len* bytes then the first *str* characters are returned. Otherwise *str* is suffixed with '0' (`#a000`) characters.

`string->key-hash` *str len hash-fun* [Bigloo Cryptography procedure]
The input string *str* is run through the given hash function *hash-fun*. The result is then concatenated multiple times (with itself) until a string of the *len* bytes is obtained.

In the following example we encrypt *some-message* using a password `"my password"`. The password will be transformed to 256 bits (32 bytes) using the `string->key256` function.

```
(define (string->key256 password)
  (string->key-hash password 32
                    (lambda (str) (string-hex-intern (sha1sum str)))))
(encrypt 'aes some-message "my password" :string->key string->key256)
```

Note that the following example yields an identical result:

```
(define (string->key256 password)
  (string->key-hash password 32
                    (lambda (str) (string-hex-intern (sha1sum str)))))
(encrypt 'aes some-message (string->key256 "my password")
         :string->key (lambda (x) x))
```

`string->key-simple` *str len hash-fun* [Bigloo Cryptography procedure]
This function implements the simple s2k algorithm of OpenPGP (RFC 2440). Basically *str* is run through the hash-fun several times until the concatenation of the results is long enough. At each iteration the string is prefixed with *count* '0'-bytes (where *count* is the iteration counter).

`string->key-salted` *str len hash-fun salt* [Bigloo Cryptography procedure]
This function implements the salted s2k algorithm of OpenPGP (RFC 2440). Similar to `string->key-simple` but the input string is first prefixed with *salt*.

`string->key-iterated-salted` *str len hash-fun salt count* [Bigloo Cryptography procedure]
This function implements the iterated salted s2k algorithm of OpenPGP (RFC 2440). The variable *count* must be a long. This algorithm is an extension of `string->key-salted` where the hash function is applied repeatedly.

This function has changed with release 3.4b. Earlier versions could be incompatible with RFC 2440.

14.2 Public Key Cryptography

14.2.1 Rivest, Shamir, and Adleman (RSA)

Bigloo's implementation of RSA is based on RFC 3447, PKCS #1 v2.1. It does not feature multiprime RSA, though.

Bigloo's implementation is *not* secure against timing attacks. Furthermore some error codes might reveal information to attackers.

14.2.1.1 RSA Keys

There are two kinds of RSA keys inside Bigloo: complete and partial keys. A complete key contains the information of both the public and the private key (together with other information that could be reconstructed out of the private key). A partial key just contains the modulus and the private or public exponent.

`RSA-Key` [Bigloo Cryptography class]

`Complete-RSA-Key` [Bigloo Cryptography class]

```
(class Rsa-Key modulus::bignum exponent::bignum)
(final-class Complete-Rsa-Key::Rsa-Key
  ;; for the complete-rsa-key "exponent" takes the role of 'd'
  e::bignum p::bignum q::bignum
  exp1::bignum   ;; d mod (p-1)
  exp2::bignum   ;; d mod (q-1)
  coeff::bignum) ;; (inverse of q) mod p
```

RSA keys can be read and written using `read-pem-key` and `write-pem-key` (Section 14.2.4 [PEM], page 164).

`generate-rsa-key` [*:key 1024*] [*:show-trace*] [Bigloo Cryptography procedure]

This function generates a new RSA key (with its public and private components).

Do not use this function for critical applications. No special effort has been undertaken to guarantee the randomness of the generated prime numbers, nor to weed out insecure keys.

Complete keys can be accessed using the following functions:

`extract-public-rsa-key` *complete-key* [Bigloo Cryptography procedure]

Returns the public partial key of the given complete key.

This procedure is implemented as follows:

```
(define (extract-public-rsa-key::Rsa-Key key::Complete-Rsa-Key)
   (with-access::Complete-Rsa-Key key (modulus e)
      (make-Rsa-Key modulus e)))
```

`extract-private-rsa-key` *complete-key* [Bigloo Cryptography procedure]

Returns the private partial key of the given complete key.

`rsa-key=?` *key1 key2* [Bigloo Cryptography procedure]

Returns true if the two keys have the same modulus and public exponent. The exponent of a partial key is considered to be public.

`rsa-key-length` *key* [Bigloo Cryptography procedure]

Returns the key length in bytes.

14.2.1.2 RSA basic operations

RSA only works on bignums (up to the size of the modulus). The following procedures implement basic encryption, decryption, signing and signature verification.

`rsa-encrypt` *key m* [Bigloo Cryptography procedure]
Encrypts the bignum *m* using the given key. If the key is a complete key then its public exponent is used. For partial keys only one exponent is available (which is assumed to be the public 'e' of the recipient). The result is again a bignum.

`rsa-decrypt` *key c* [Bigloo Cryptography procedure]
Decrypts the bignum *c* using the given key. If the key is a complete key then its private exponent is used. For partial keys only one exponent is available (which is assumed to be the private 'd'). The result is again a bignum.

`rsa-sign` *k m* [Bigloo Cryptography procedure]
Signs the bignum *m* using key *k*. Uses the private exponent of complete keys. The result is a bignum.

`rsa-verify` *k m s* [Bigloo Cryptography procedure]
Verifies the signature *s*. Returns true if *s* is the signature of *m*. The key *k* should be the public key of the signer.

14.2.1.3 Examples

In this section we will present an example of using RSA.

Let's start by generating an RSA key in openssl:

```
$ openssl genrsa -out my_rsa_key.pem 1024
```

Our key will have 1024 bits (for the public modulus), and therefore RSA will only be able to work with bignums up to 1024 bits (128 bytes).

Now some Bigloo code that uses this key.

Start by loading the library.

```
(module rsa-example (library crypto))
```

Now read the key:

```
(define *key* (read-pem-key "my_rsa_key.pem"))
(define *public-key* (extract-public-rsa-key *key*))
```

The public portion of the key can be distributed:

```
;; publish the *public-key*:
(write-pem-key-string *public-key*)
```

Now let's sign the message "My Important Message". This message is sufficiently short to be signed directly, but in general it is better to get a hash of the message:

```
(define msg-hash (sha1sum "my message"))
(define msg-hash-bignum (octet-string->bignum msg-hash))
```

The result of `sha1sum` returns a human readable representation of the hash. It would hence be possible to transform it back to an internal representation before applying the `octet-string->bignum` function:

```
(define msg-hash-bignum (octet-string->bignum (string-hex-intern msg-hash)))
```

In our case both variants are small enough to fit into our keys. The latter version is however more often used.

Now that we have a message hash in bignum form we can sign it.

```
(define signature (rsa-sign *key* msg-hash-bignum))
```

The signature is again in bignum form. If needed there are several ways to transform it into string-form (for instance `bignum->string` or `bignum->octet-string`).

The signature can now be distributed. Anyone wanting to verify the signature simply has to create the same message-hash and call `rsa-verify` with our public key:

```
(rsa-verify *public-key* msg-hash-bignum signature) ⇒ #t
```

Encryption and decryption work in a similar way.

Suppose someone (let's say "Alice") wants to send us the following secret message "Cryptography". The encryption and decryption functions work, similar to the signature functions, on bignums. We could, as before, simply transform this short string into a bignum and directly encrypt the bignum. This approach would however not work for longer strings. In the following we will present the generic version that works with strings of any size.

Public key cryptography is relatively slow and Alice thus starts by encrypting our message a fast block cipher with a "random" password:

```
(define encrypted (encrypt 'aes "Cryptography" "my random password"))
```

Alice can already send us the encrypted message. We will just not yet be able to decrypt it, as we don't have the random password yet.

Alice now takes her random password string and encrypts it with our public key:

```
(define encrypted-key (rsa-encrypt *public-key* (octet-string->bignum "my random password")))
```

Alice simply sends us the `encrypted-key`. On our side we can now decrypt the key:

```
(define aes-key (bignum->octet-string (rsa-decrypt *key* encrypted-key)))
```

We can now decrypt the previously received message:

```
(decrypt 'aes aes-key encrypted) ⇒ "Cryptography"
```

14.2.1.4 RSA RFC 3447

The following functions have been defined in RFC 3447.

`RSAEP` *k m* [Bigloo Cryptography procedure]
`RSADP` *k c* [Bigloo Cryptography procedure]
`RSASP1` *k m* [Bigloo Cryptography procedure]
`RSAVP1` *k s* [Bigloo Cryptography procedure]

These are the RFC 3447 names for encryption, decryption, signature and signature verification. Note that the verification does not receive the original message as parameter.

In fact `rsa-verify` is implemented as follows:

```
(define (rsa-verify k m s)
  (=bx m (RSAVP1 k s)))
```

`PKCS1-v1.5-pad` *m-str key-len mode* [Bigloo Cryptography procedure]
`PKCS1-v1.5-unpad` *em-str mode* [Bigloo Cryptography procedure]

Pads (resp. unpads) the given string using PKCS1-v1.5 specifications. Mode must be 0, 1 or 2.

`RSAES-PKCS1-v1.5-encrypt` *key m-str* [Bigloo Cryptography procedure]
`RSAES-PKCS1-v1.5-decrypt` *key c-str* [Bigloo Cryptography procedure]
`RSASSA-PKCS1-v1.5-sign` *key msg-str* [Bigloo Cryptography procedure]
[*:hash-algo 'sha-1*]
`RSASSA-PKCS1-v1.5-verify` *key msg-str S-str* [Bigloo Cryptography procedure]
`RSASSA-PKCS1-v1.5-sign-bignum` *key msg-str* [Bigloo Cryptography procedure]
[*:hash-algo 'sha-1*]
`RSASSA-PKCS1-v1.5-verify-bignum` *key* [Bigloo Cryptography procedure]
msg-str S

RSAES-PKCS1-v1.5 functions work on strings. However their length is limited by the size of the modulus (to be exact: by key-len - 11). The `-bignum` functions skip the last step of converting the internal bignum to strings.

The optional `:hash-algo` must be either `sha-1` or `md5` (RFC 3447 allows other hash algorithms, but they are not yet implemented).

`RSAES-OAEP-encrypt` *key m-str* [*:label* `""`] [Bigloo Cryptography procedure]
`RSAES-OAEP-decrypt` *key cypher-str* [*:label* `""`] [Bigloo Cryptography procedure]
`RSASSA-PSS-sign` *key msg-str* [Bigloo Cryptography procedure]
`RSASSA-PSS-verify` *key msg-str sig-str* [Bigloo Cryptography procedure]

These functions pad, mask, etc the input string before they perform their operation on them. See RFC 3447 for more information.

14.2.2 Digital Signature Algorithm (DSA)

Bigloo has rudimentary (but usually sufficient) support for DSA. While it is not possible to generate new DSA keys inside Bigloo one can sign or verify with Bigloo.

DSA keys can be read and written using `read-pem` (Section 14.2.4 [PEM], page 164).

For consistency with RSA we have named DSA keys in a similar way as the RSA keys. The public part of a DSA key can be found in the class `DSA-Key` while the private part is added in the `Complete-DSA-Key` subclass.

`DSA-Key` [Bigloo Cryptography class]
`Complete-DSA-Key` [Bigloo Cryptography class]

```
(class Dsa-Key
   p::bignum q::bignum g::bignum y::bignum)
(final-class Complete-Dsa-Key::Dsa-Key
   x::bignum)) ;; the private key
```

`extract-public-dsa-key` *complete-key* [Bigloo Cryptography procedure]

Returns a `DSA-Key` without the private *x*.

`dsa-sign` *m key* [Bigloo Cryptography procedure]

Signs the bignum *m* using the private dsa key *key*. The result are two values: *r* and *s*.

A typical call to `dsa-sign` is hence of the following form

```
(receive (r s)
  (dsa-sign secret-key hashed-msg-bignum)
  (process-signature r s))
```

`dsa-verify` *m key r s* [Bigloo Cryptography procedure]
Verifies a signature (consisting of *r* and *s*).

DSA works very similar to RSA. Have a look at RSA's example section.

14.2.3 ElGamal

Bigloo supports ElGamal encryption (but not signing). Bigloo's implementation is minimal.

For consistency with RSA ElGamal keys are similarly named as their RSA counterparts.

`ElGamal-Key` [Bigloo Cryptography class]
`Complete-ElGamal-Key` [Bigloo Cryptography class]

```
(class ElGamal-Key
  p::bignum
  g::bignum
  y::bignum)
(final-class Complete-ElGamal-Key::ElGamal-Key
  x::bignum)) ;; the private key
```

`extract-public-elgamal-key` *complete-key* [Bigloo Cryptography procedure]
Returns a copy of the public part (as `ElGamal Key`).

`elgamal-encrypt` *key m* [Bigloo Cryptography procedure]
Encrypts the bignum *m* using the given public key. The result are two values `c1` and `c2`.

Note that ElGamal encryption needs random bytes for every encryption. This means that this function may return different results with the same parameters. It furthermore implies that the result is insecure if the operating system provides bad random numbers, or if Bigloo's random-number generation is buggy. For critical applications be sure to verify both requirements.

`elgamal-decrypt` *complete-key c1 c2* [Bigloo Cryptography procedure]
Decrypts an ElGamal encrypted message (consisting of the two bignums *c1* and *c2*) with the given private key.

`elgamal-key-length` *key* [Bigloo Cryptography procedure]
Returns the key length in bytes.

ElGamal works very similar to RSA. Have a look at RSA's example section.

14.2.4 PEM

Bigloo is able to read and write RSA and DSA keys in PEM format. This is the default format used by OpenSSL.

The following example creates a new DSA key pair in OpenSSL and stores it in PEM format.

```
$ openssl dsaparam 1024 -out /tmp/dsaparam
$ openssl gendsa /tmp/dsaparam
```

`read-pem-key` *in* [Bigloo Cryptography procedure]
`read-pem-key-port` *input-port* [Bigloo Cryptography procedure]
`read-pem-key-file` *filename* [Bigloo Cryptography procedure]

`read-pem-key-string` *str* [Bigloo Cryptography procedure]

These functions will read a PEM encoded key. The encoded file may contain a private or public RSA key, or a private or public DSA key.

The procedure `read-pem-key` accepts input-ports and strings. In the case of a string it will invoke `read-pem-key-file` (and not `read-pem-key-string`).

`write-pem-key` *key out* [*public-key-only?*] [Bigloo Cryptography procedure]
`write-pem-key-port` *key out* [*public-key-only?*] [Bigloo Cryptography procedure]
`write-pem-key-file` *key out* [*public-key-only?*] [Bigloo Cryptography procedure]
`write-pem-key-string` *key* [*public-key-only?*] [Bigloo Cryptography procedure]

These functions write the given key. The key may be a private/public RSA/DSA key.

The procedure `write-pem-key` accepts output-ports and strings as *out* parameter. If *out* is a string it will delegate to `write-pem-key-file`.

14.3 OpenPGP

Bigloo implements parts of OpenPGP (RFC 2440, RFC 4880). All OpenPGP functions are accessible via the `openpgp` library.

Here is an example of a module that uses this library:

```
;; Encrypt a string using openpgp default encryption.
(module pgp-encrypt
   (library openpgp)
   (main main))

(define (main argv)
   (when (and (pair? (cdr argv)) (pair? (cddr argv)))
      (let ((encrypt? (string=? "-e" (cadr argv)))
            (passwd (caddr argv))
            (input (read-string)))
         (if encrypt?
             (display (pgp-write-string (pgp-encrypt input
                                                     '()  ;; no public keys
                                                     (list passwd))))
             (let ((composition (pgp-read-string input)))
               (display (pgp-decrypt composition
                                     :passkey-provider (lambda () passwd))))))))
```

`pgp-read-string` *str* [Bigloo OpenPGP procedure]
`pgp-read-port` *iport* [Bigloo OpenPGP procedure]
`pgp-read-file` *file-name* [Bigloo OpenPGP procedure]

These functions read and decode PGP data. OpenPGP allows several keys to be stored in the same message. Therefore `pgp-read` will return keys always in a list (even if the message only contains one key).

The return value is either a list of PGP-compositions (PGP-Keys), or a single PGP-composition.

`pgp-write-string` *composition* [*:format 'armored*] [Bigloo OpenPGP procedure]
`pgp-write-port` *oport composition* [*:format 'armored*] [Bigloo OpenPGP procedure]

`pgp-write-file` *file-name composition* [*:format 'armored*] [Bigloo OpenPGP procedure]

The counter-part of `pgp-read`. These functions encode PGP-compositions. By default the result is armored (i.e. encoded with ASCII characters). If the optional `:format` parameter is different than the symbol `armored`, then the composition is encoded in binary.

Note that there is no means to encode a list of PGP-keys.

`pgp-encrypt` *msg-string keys passwords* [*:hash-algo 'sha-1*] [*:symmetric-algo 'cast5*] [Bigloo OpenPGP procedure]

Encrypts the given string. The returned composition can be decrypted by the owners of the keys, or with one of the passwords.

In the following example Alice and Bob may use their private key to decrypt the secret message. Users knowing the one of the passwords ("foo" and "bar") will also be able to decrypt the message.

```
(pgp-write-file "encrypted.pgp"
   (pgp-encrypt "my secret message"
                (list alice-key bob-key)
                '("foo" "bar")))
```

The given keys should be subkeys of a PGP-key, but if a PGP-key is given Bigloo will do its best to pick the correct subkey for encryption.

- If only one subkey exists (the main-key) then this subkey is used.
- If two subkeys exist, and the non-main key is suitable for encryption, then the non-main key is used.
- If only one of many subkeys (including the main-key) is suitable for encryption, then this subkey is used.
- Else Bigloo raises an error.

`pgp-password-encrypt` *msg-string password* [*:hash-algo 'sha-1*] [*:symmetric-algo 'cast5*] [*:mdc #t*] [Bigloo OpenPGP procedure]

Deprecated. Encrypts `msg-string` with the given password. The returned PGP-composition does not contain any information which hash-algorithm and symmetric encryption algorithm has been used. RFC 4880 specifies that IDEA and MD5 should be used. However GnuPG uses SHA-1 and CAST5. Therefore Bigloo defaults to the latter algorithms.

Even though the usage of this PGP message is deprecated it yields the smallest encrypted data. It may be of interest when compatibility with other tools is not a requirement (but why use OpenPGP then).

The optional `mdc` flag triggers the usage of a modification detection code. It is more secure against tampering but requires more space and might not be recognized by old openpgp implementations.

`pgp-decrypt` *encrypted* [*:passkey-provider* (*lambda* () *#f*)] [*:password-provider* (*lambda* (*key*) *#f*)] [*:key-manager* (*lambda* (*key-id*) *'()*)] [*:hash-algo 'sha-1*] [*:symmetric-algo 'cast5*] [Bigloo OpenPGP procedure]

Decrypts a PGP-composition that has been generated by `pgp-encrypt` or by `pgp-password-encrypt`. The function returns the decrypted message (a string) or `#f` if decryption was not possible.

If the message can be decrypted with a private key, then Bigloo will call the `key-manager` and request a list of PGP-subkeys that match the given key-id.

If a subkey (returned by the key-manager) is not yet decrypted, Bigloo will invoke the `password-provider` with the subkey, and request a password to decrypt the private part of the subkey.

If the message can be decrypted with a password Bigloo will then request a passkey by invoking the `passkey-provider`.

The optional arguments `hash-algo` and `symmetric-algo` are only used for messages that have been encrypted with `pgp-password-encrypt`.

`pgp-sign` *msg-string key password-provider* [*:detached-signature? #t*] [*:one-pass? #f*] [*:hash-algo 'sha-1*] [Bigloo OpenPGP procedure]

Signs `msg-string` with the given key. Ideally the key should be a subkey, but if a complete PGP-Key is given, Bigloo will use the main-key instead. If the main-key is not suitable for signing, then an error is raised.

If the private part of the key has not yet been decrypted then Bigloo will call the password-provider (a procedure) with the subkey to get a password (a string).

The function returns a PGP-composition.

If the optional `detached-signature?` parameter is set to `#f` then the msg-string is not included in the returned composition.

The `one-pass?` and `hash-algo` parameters are usually left at its default values.

Example:

```
(let ((my-key (car (pgp-read-file "my-key.pgp"))))
  (pgp-write-file "msg.sig"
     (pgp-sign "my signed message"
               my-key
               (lambda (key) "my-password")
               :detached-signature? #f)))
```

`pgp-verify` *signature key-manager* [*:msg #f*] [Bigloo OpenPGP procedure]

Verifies a signature.

The key-manager is a function that takes a substring identifier and returns a list of keys matching this id. Since a signature composition may contain several signatures this function may be invoked several times.

The result is a list of subkeys that signed the message. If the key-manager doesn't have any of the signature-keys then the result is the empty list.

A message (string) needs only be given if the signature is detached. Otherwise the original message is encoded in the signature-composition.

Example:

```
(let ((sig (pgp-read-file "msg.sig")))
```

```
(let ((signers (pgp-verify sig my-key-manager)))
 (for-each (lambda (subkey)
             (print (subkey->string subkey) " signed the message"))
           signers)))
```

`pgp-signature-message` *signature* [Bigloo OpenPGP procedure]
Returns the signature's message, or `#f` if the signature is a detached signature.

`pgp-key?` *key* [Bigloo OpenPGP procedure]
`pgp-subkey?` *key* [Bigloo OpenPGP procedure]
Predicates for PGP-Key and PGP-Subkey.

`pgp-subkeys` *key* [Bigloo OpenPGP procedure]
Returns a list of PGP-Subkeys of the PGP-Key. The first key in the list is the *main-key*. The main-key is used as default for signatures.

`pgp-key->string` *key* [Bigloo OpenPGP procedure]
`pgp-subkey->string` *key* [Bigloo OpenPGP procedure]
Returns a string representation of the key (resp. subkey).

Example outputs:

```
(pgp-key->string key)
⇒ John Doe john.doe@gmail.com
⇒ bd4df3b2ddef790c RSA (Encrypt or Sign)
⇒ 424610a65032c42e RSA (Encrypt or Sign)

(pgp-subkey->string (car (pgp-subkeys key)))
⇒ John Doe john.doe@gmail.com
⇒ bd4df3b2ddef790c RSA (Encrypt or Sign)
```

`pgp-key-id` *subkey* [Bigloo OpenPGP procedure]
`pgp-key-fingerprint` *subkey* [Bigloo OpenPGP procedure]
Returns the id (resp. fingerprint) of a subkey.

A subkey-id is a 8-character binary string.

A fingerprint is a 20-character binary string.

`pgp-make-key-db` [Bigloo OpenPGP procedure]
`pgp-add-key-to-db` *db key* [Bigloo OpenPGP procedure]
`pgp-add-keys-to-db` *db keys* [Bigloo OpenPGP procedure]
`pgp-resolve-key` *db id* [Bigloo OpenPGP procedure]
`pgp-db-print-keys` *db* [Bigloo OpenPGP procedure]
A simple key-manager implementation based on lists.

14.3.1 Examples

14.3.1.1 Signatures

Unless you already have a gpg key create a new PGP key with gpg. Note that DSA with a keysize greater than 1024 does not work with SHA-1. SHA-224,256,384,512 would work, but are not yet implemented in Bigloo.

$ gpg –gen-key

...

```
pub   1024D/A2DA694E 2010-08-07 [expires: 2010-08-27]
      Key fingerprint = DFAF 5894 9003 8640 D45B  6199 07CA 0495 A2DA 694E
uid                  Bigloo Example
sub   1024g/0B8985E5 2010-08-07 [expires: 2010-08-27]
```

We export both the public and the private key.

```
$ gpg  -a -o A8453FAB_Bigloo_Example_User.pkey –export A8453FAB
$ gpg  -a -o A8453FAB_Bigloo_Example_User.skey –export-secret-keys A8453FAB
```

This small program will simply read the key and print a human-readable representation.

```
;; contents of print-key.scm
(module print-key
   (library openpgp)
   (main my-main))

(define (my-main args)
   (let ((public-key (car (pgp-read-file "A2DA694E_Bigloo_Example.pkey")))
         (secret-key (car (pgp-read-file "A2DA694E_Bigloo_Example.skey"))))
      (display (pgp-key->string public-key))
      (display (pgp-key->string secret-key))))
```

The compilation is straight-forward and does not require any special flags:

```
$ bigloo print-key.scm -o print-key
$ ./print-key
Bigloo Example
07ca0495a2da694e DSA (Digital Signature Standard)
5fa4e8c90b8985e5 ElGamal (Encrypt-Only)
Bigloo Example
07ca0495a2da694e DSA (Digital Signature Standard)
5fa4e8c90b8985e5 ElGamal (Encrypt-Only)
```

As can be seen, the `pgp-key->string` routine does not differentiate between public and private keys.

We can also sign a message:

```
(let ((my-key (car (pgp-read-file "A2DA694E_Bigloo_Example.skey"))))
  (pgp-write-file "msg.sig"
     (pgp-sign (read-string)
               my-key
               (lambda (key) "<Bigloo Example Password>")
               :detached-signature? #f)))
```

Signatures from Bigloo follow RFC 4880 and can therefore be verified by `gpg`.

```
$ echo "Gpg can verify Bigloo's signature" | ./sign
$ gpg –verify msg.sig
gpg: Signature made Sat 07 Aug 2010 10:12:21 PM CEST using DSA key ID A2DA694E
gpg: Good signature from "Bigloo Example"
```

Inversely Bigloo can verify `pgp`'s signature. Here we first generate a signature with `gpg`.

```
$ echo "Bigloo can verify gpg's signatures." | \
 gpg -o msg_gpg.sig -a \
    –default-key "Bigloo Example" \
    –passphrase <Bigloo Example Password> \
    –sign
```

You need a passphrase to unlock the secret key for
user: "Bigloo Example"
1024-bit DSA key, ID A2DA694E, created 2010-08-07

The following program reads OpenPGP signatures and verifies them. For simplicity the key database will only contain one key, but it could contain any number of keys.

```
(let ((my-key (car (pgp-read-file "A2DA694E_Bigloo_Example.pkey")))
      (sig (pgp-read-file "msg_gpg.sig"))
      (db (pgp-make-key-db)))
   (pgp-add-key-to-db db my-key)
   (print "Signature message: " (pgp-signature-message sig))
   (let ((signers (pgp-verify sig (lambda (id) (pgp-resolve-key db id)))))
      (for-each (lambda (subkey)
                  (display (pgp-subkey->string subkey)))
                signers)))
```

As expected, the program verifies the correct signature.

$ ./verify
Signature message: Bigloo can verify gpg's signatures.

Bigloo Example
07ca0495a2da694e DSA (Digital Signature Standard)

14.3.1.2 Email Usage

Usage of OpenPGP within mails is described in RFC 3156.

Encrypted parts and signatures are encoded with their separate content-types. Signatures are done over a canonicalized version of the message. They also hash over the content-type headers.

OpenPGP's recette program has an example for a signature from kmail, that can be succesfully verified with Bigloo.

14.3.1.3 Encryption

OpenPGP allows messages to be encrypted with passwords (in this context "passkey") or public keys. It is also possible to encrypt a message for more than one recipient. In such a case the data will be encrypted by a session-key which in turn is encrypted separately for each recipient. Since the session-key is not very big (compared to the data) the size overhead is usually insignificant.

Let's start by encrypting a message with a simple passkey.

```
(let* ((secret-data "My secret data\n")
       (composition (pgp-encrypt secret-data '() '("My secret passkey"))))
  (pgp-write-file "secret.pgp" composition))
```

As usual the pgp message is compatible with gpg:

$ gpg secret.pgp
gpg: CAST5 encrypted data
Enter passphrase: <My secret passkey>
gpg: encrypted with 1 passphrase
$ cat secret
My secret data

As expected, Bigloo can decrypt password protected files that have been generated by gpg:

```
$ echo "A secret message encrypted with gpg." | \
  gpg -o encrypted.pgp –symmetric \
     –passphrase "secret key"
```

The Bigloo code to decrypt the message is very simple:

```
(print (pgp-decrypt (pgp-read-file "encrypted.pgp")
                    :passkey-provider (lambda () "secret key")))
```

In a similar vein it is possible to use public key encryption. The following example tests the encryption and decryption part of Bigloo.

```
(let* ((my-key (car (pgp-read-file "A2DA694E_Bigloo_Example.skey")))
       (db (pgp-make-key-db))
       (secret-data "My secret message")
       (encrypted (pgp-encrypt secret-data `(,my-key) '())))
   (pgp-add-key-to-db db my-key)
   (let* ((key-manager (lambda (id) (pgp-resolve-key db id)))
          (password-provider (lambda (key) <Bigloo Example Password>))
          (decrypted (pgp-decrypt encrypted
                                  :key-manager key-manager
                                  :password-provider password-provider)))
     (if (not (string=? decrypted secret-data))
         (error "decrypt-test"
                "Something went horribly wrong"
                decrypted))))
```

Note that a secret secret key has a part that is encrypted by a password. During decryption Bigloo needs access to this encrypted data and therefore invokes the password-provider so it can decrypt it. In many cases this will trigger an interactive callback with the user. Here, in this toy example, we know that the password that is needed is for the Bigloo Example key. In a more general case the password-provider will have to print the key to give more information to the user.

In the following example we show how to encrypt data for 3 passwords and one key.

```
(let* ((my-key (car (pgp-read-file "A2DA694E_Bigloo_Example.skey")))
       (db (pgp-make-key-db))
       (secret-data "My secret message")
       (encrypted (pgp-encrypt secret-data `(,my-key)
                                           '("pass1" "pass2" "pass3"))))
   (pgp-write-file "multi_receiver.pgp" encrypted))
```

We believe that gpg has a bug and does not know how to handle such messages correctly. Bigloo, however, decrypts the message with any of the possible options.

14.4 Development

Bigloo's OpenPGP implementation only exposes few library functions. As a consequence some features are not accessible. The key-management system is very rough, and there are no means to inspect messages in more detail. It should be possible to expose or implement many of those missing features with little effort. The most time-consuming part is generally designing a clean API and the testing/debugging of new features: when something goes wrong it can take a huge amount of time to find the reason.

Developers interested in improving Bigloo's OpenPGP library can print a huge amount of debugging information by enabling the debug-macro in `util.scm`. Bigloo's OpenPGP

implementation is not designed for speed and takes no shortcuts. The debugging output can therefore be used to follow the specification of RFC 4880 (or 2440).

15 Errors, Assertions, and Traces

15.1 Errors and Warnings

Bigloo permits to signal an error via the `error` function. Errors are implemented by the means of exceptions (see `with-exception-handler`, `with-handler`, and `raise` forms). Assertions allow the checking of predicates at certain points in programs.

`typeof` *obj* [bigloo procedure]

Returns a string which is the name of the dynamic type of *obj*.

`error` *proc msg obj* [bigloo procedure]

This form signals an error by calling the current error handler with *proc*, *msg* and *obj* as arguments.

```
(define (foo l)
   (if (not (pair? l))
       (error "foo" "argument not a pair" l)
       (car l)))

(foo 4)
error  *** ERROR:bigloo:foo:
        argument not a pair -- 4
```

Switching on the `-g` compilation switch enables stack dumping when the `error` function is invoked. That is, when a program is compiled with `-g` and when, at runtime, the shell variable `BIGLOOSTACKDEPTH` is set and contains a number, an execution stack of depth `BIGLOOSTACKDEPTH` is printed when an error is raised.

`error/location` *proc msg obj file location* [bigloo procedure]

This form signals an error by calling the current error handler with *proc*, *msg* and *obj* as arguments. The error is prompted in *file*, at character position *location*.

```
(define (foo l)
   (if (not (pair? l))
       (error/location
         "foo" "argument not a pair" l "foo.scm" 115)
       (car l)))

(foo 4)
error  File "foo.scm", line 4, character 115:
        #       (car l)))
        #       ^
        # *** ERROR:bigloo:foo
        # argument not a pair -- 4
            0. FOO
            1. DYNAMIC-WIND
            2. INTERP
            3. ENGINE
            4. MAIN
```

`get-trace-stack` *size* [bigloo procedure]

`dump-trace-stack` *output-port size* [bigloo procedure]

Switching on the `-g` compilation switch enables stack dumping Chapter 31 [Compiler Description], page 275. That is, the list of the pending calls can be dumped by the

runtime-system. The function `get-trace-stack` builds such a trace. The list built by `get-trace-stack` only contains the *size* top most pending calls. The function `dump-trace-stack` displays a representation of this stack on the *output-port*.

`warning/location` *file location* [*arg*]... [bigloo procedure]
This form signals a warning. That is, is *arg* are displayed on the standard error port. The warning is prompted in *file* at character position *location*.

```
(define (foo l)
   (if (not (pair? l))
       (begin
          (warning/location
            "foo.scm" 154 "foo:" "argument not a pair -- " l)
          '())
       (car l)))

(foo 4)
⊣ File "foo.scm", line 6, character 154:
   #       (car l)))
   #       ^
   # *** WARNING:bigloo:foo:
   argument not a pair -- 4
⇒ '()
```

`exception-notify` *exc* [bigloo procedure]
`error-notify` *err* [bigloo procedure]
`warning-notify` *err* [bigloo procedure]
Display a message describing the error or warning on the default error port.

15.2 Exceptions

`current-exception-handler` [SRFI-18 function]
Returns the current exception handler with is a 0-ary procedure.

`with-exception-handler` *handler thunk* [SRFI-18 function]
Returns the result(s) of calling *thunk* with no arguments. The *handler*, which must be a procedure, is installed as the current exception handler in the dynamic environment in effect during the call to *thunk*. When possible, prefer `with-handler` to `with-exception-handler` because the former provides better debugging support and because its semantics is more intuitive.

`with-handler` *handler body* [bigloo form]
Returns the result(s) of evaluating *body*. The *handler*, which must be a procedure, is installed as the current exception handler in the dynamic environment in effect during the evaluation of *body*. Contrarily to `with-exception-handler`, if an exception is raised, the *handler* is invoked and the value of the `with-handler` form is the value produced by invoking the *handler*. The handler is executed in the continuation of the `with-handler` form.

JVM note: When executed within a JVM, the form `with-handler` also catches Java exceptions.

Important note: Since Bigloo version 3.2c, error handlers are executed *after* the execution stack is unwound. Hence, error handlers are executed *after* protected blocks. For instance in the following code:

```
(with-handler
   (lambda (e) action)
   (unwind-protect
      body
      protect))
```

The *action* is executed *after protect*.

`raise` *obj* [SRFI-18 function]

Calls the current exception handler with obj as the single argument. obj may be any Scheme object. Note that invoking the current handler does not escape from the current computation. It is up the to handler to perform the escape. It an error, signaled by the runtime system, if the current exception handler returns.

```
(define (f n)
  (if (< n 0) (raise "negative arg") (sqrt n))))

(define (g)
  (bind-exit (return)
    (with-exception-handler
      (lambda (exc)
        (return
          (if (string? exc)
              (string-append "error: " exc)
              "unknown error")))
      (lambda ()
        (write (f 4.))
        (write (f -1.))
        (write (f 9.))))))

(g)  ⊣ 2. and returns "error: negative arg"
```

The standard Bigloo runtime system uses the following classes for signaling errors and warnings:

- `&exception` which is defined as:

```
(class &exception
   (fname read-only (default #f))
   (location read-only (default #f)))
```

- `&error` defined as:

```
(class &error::&exception
   (proc read-only)
   (msg read-only)
   (obj read-only))
```

- `&type-error` defined as:

```
(class &type-error::&error
   (type read-only))
```

- `&io-error` defined as:

```
(class &io-error::&error)
```

- `&io-port-error` defined as:

```
(class &io-port-error::&io-error)
```

- `&io-read-error` defined as:
  ```
  (class &io-read-error::&io-port-error)
  ```
- `&io-write-error` defined as:
  ```
  (class &io-write-error::&io-port-error)
  ```
- `&io-closed-error` defined as:
  ```
  (class &io-closed-error::&io-port-error)
  ```
- `&io-file-not-found-error` defined as:
  ```
  (class &io-file-not-found-error::&io-error)
  ```
- `&io-parse-error` defined as:
  ```
  (class &io-parse-error::&io-error)
  ```
- `&io-unknown-host-error` defined as:
  ```
  (class &io-unknown-host-error::&io-error)
  ```
- `&io-malformed-url-error` defined as:
  ```
  (class &io-malformed-url-error::&io-error)
  ```
- `&http-error` defined as:
  ```
  (class &http-error::&error)
  ```
- `&http-redirection-error` defined as:
  ```
  (class &http-redirection-error::&http-error)
  ```
- `&http-status-error` defined as:
  ```
  (class &http-status-error::&http-error)
  ```
- `&http-redirection` defined as:
  ```
  (class &http-redirection::&exception
    (port::input-port read-only)
    (url::bstring read-only))
  ```
- `&process-exception` defined as:
  ```
  (class &process-exception::&error)
  ```
- `&warning` defined as:
  ```
  (class &warning::&exception
     (args read-only))
  ```
- `&eval-warning` defined as:
  ```
  (class &warning::&warning)
  ```

15.3 Deprecated try form

`try` *exp handler* [bigloo syntax]

This form is deprecated. As much as possible, it should be replaced with true exceptions (i.e., `with-exception-handler` and `raise`). The argument *exp* is evaluated. If an error is raised, the *handler* is called. The argument *handler* is a procedure of four arguments. Its first argument is the continuation of *try*. The other arguments are *proc*, *mes* and *obj*. Invoking the first argument will resume after the error.

```
(let ((handler (lambda (escape proc mes obj)
                  (print "***ERROR:" proc ":" mes " -- " obj)
                  (escape #f))))
   (try (car 1) handler))
   ⊣ ***ERROR:CAR:not a pair -- 1
```

```
⇒ #f
```

The argument *handler* is not evaluated in the dynamic scope of its `try` form. That is:

```
(let ((handler (lambda (escape proc mes obj)
                  (escape (car obj)))))
   (try (car 1) handler))
   error  *** ERROR:bigloo:CAR
          Type 'PAIR' expected, 'BINT' provided -- 1
```

Some library functions exist to help in writing handlers:

`warning` [*arg*]... [bigloo procedure]

This form signals a warning. That is, is *arg* are displayed on the standard error port.

```
(define (foo l)
   (if (not (pair? l))
       (begin
          (warning "foo:" "argument not a pair -- " l)
          '())
       (car l)))

(foo 4)
⊣ *** WARNING:bigloo:foo:
  argument not a pair -- 4
⇒ '()
```

15.4 Assertions

`assert` (*var*...) *s-expression* [bigloo syntax]

Assertions can be enabled or disabled using Bigloo's compilation flags `-g` flag to enable them). If the assertions are disabled they are not evaluated. If an assertion is evaluated, if the expression *exp* does not evaluate to `#t`, an error is signaled and the interpreter is launched in an environment where *var*... are bound to their current values.

Assertion forms are legal expressions which *always* evaluate to the `unspecified` object.

Here is an example of assertion usage:

```
(module foo
   (eval (export foo)))

(define (foo x y)
   [assert (x y) (< x y)]
   (labels ((gee (t)
               [assert (t) (>= t 0)]
               (let ((res (+ x t)))
                  [assert (res t) (> res 10)]
                  res)))
      (set! x (gee y))
      [assert (x) (> x 10)]
      x))

(repl)
```

This module is compiled with the `-g` flag to enable assertions, then the produced executable is run:

```
$ a.out

1:=> (foo 1 2)

File "foo.scm", line 9, character 158:
#                  [assert (res t) (> res 10)]
#                  ^
# *** ERROR:bigloo:assert
# assertion failed – (BEGIN (> RES 10))
   0. GEE
   1. FOO
_____________
Variables' value are :
  RES : 3
  T : 2
_____________
*:=> ^D
File "foo.scm", line 12, character 228:
#      [assert (x) (> x 10)]
#      ^
# *** ERROR:bigloo:assert
# assertion failed – (BEGIN (> X 10))
   0. FOO
_____________
Variables' value are :
  X : 3
_____________

*:=> 3
1:=> (foo 1 2)
File "foo.scm", line 9, character 158:
#                  [assert (res t) (> res 10)]
#                  ^
# *** ERROR:bigloo:assert
# assertion failed – (BEGIN (> RES 10))
   0. GEE
   1. FOO
_____________
Variables' value are :
  RES : 3
  T : 2
_____________

*:=>
```

15.5 Tracing

Bigloo provides a trace facility whose is intended for simple debugging tasks. It is a replacement for user `display`s that clutters the source code. Here is a typical example using it:

```
(define (foo x)
   (with-trace 1 'foo
      (let loop ((n x))
 (with-trace 2 'loop
    (trace-item "n=" n)
    (when (> n 0)
       (let liip ((m n))
  (with-trace 2 'liip
     (trace-item "m=" m))
  (when (> m 0)
     (liip (- m 1))))
       (loop (- n 1)))))))

(foo 3)
```

which produces the following output:

```
+ foo
  |--+ loop
  |  |- n=3
  |  |--+ liip
  |  |  |- m=3
  |  |--+ liip
  |  |  |- m=2
  |  |--+ liip
  |  |  |- m=1
  |  |--+ liip
  |  |  |- m=0
  |  |--+ loop
  |  |  |- n=2
  |  |  |--+ liip
  |  |  |  |- m=2
  |  |  |--+ liip
  |  |  |  |- m=1
  |  |  |--+ liip
  |  |  |  |- m=0
  |  |  |--+ loop
  |  |  |  |- n=1
  |  |  |  |--+ liip
  |  |  |  |  |- m=1
  |  |  |  |--+ liip
  |  |  |  |  |- m=0
  |  |  |  |--+ loop
  |  |  |  |  |- n=0
```

Traces generation is controlled by a set of functions and parameters (see Chapter 24 [Parameters], page 231). The functions are described in this chapter.

`with-trace` *level label . body* [bigloo syntax]

The variable *level* is the level of a trace. It is a positive integer. It enables simple filtering for traces. A trace is displayed if and only if the debugging level used to compile or to execute the program is greater than the trace level. The variable *label* is a label, .e.i., an identifier denoting the trace. This identifier will be displayed in debug mode. The variable *body* is the body of the form, that is, the expression to be evaluated.

Unless a trace is activated `(with-trace lv la body)` (when its level *lv* is greater than the current debug level) is equivalent to `(begin body)`. When traces are activated, before executing *body*.

The debugging level is controlled by two parameters: `bigloo-debug` and `bigloo-compiler-debug` (see Chapter 24 [Parameters], page 231).

`trace-item` *. args* [bigloo function]

This function displays all its arguments. It has to be used nested in a *with-trace* form.

`trace-bold` *s* [bigloo function]
`trace-string` *s* [bigloo function]

These two functions are provided for convenience. They returns strings made of their parameters.

`trace-color` *color . args* [bigloo function]

The *color* argument is a positive integer. This function returns a string which is the representation of *args* and that appears on the terminal in color *color*.

Colors can be enable or disabled using the `bigloo-trace-color` parameter (see Chapter 24 [Parameters], page 231).

`trace-margin` [bigloo function]
`trace-margin-set!` [bigloo function]

The *trace-margin* parameter is used to control the characters that are displayed in the margin of a trace. Usual applications should not use this. However, it may be convenient to set the margin by hands in some context. For instance, it can be used to distinguished threads in a multi-threaded application such as:

```
(make-thread (lambda ()
                (trace-margin-set! (trace-color 1 "="))
                ...))
(make-thread (lambda ()
                (trace-margin-set! (trace-color 2 "="))
                ...))
```

`trace-port` [bigloo function]
`trace-port-set!` [bigloo function]

These functions return and set the output port used by traces.

16 Threads

Bigloo supports multithreaded programming. Two different libraries programming are available. The first one, the *Fair Thread* (see Section Section 16.2 [Fair Threads], page 185), enables, simple, easy to develop and to maintain code. The second one, the Posix Thread (see Section Section 16.3 [Posix Threads], page 193) enables more easily to take benefit of the actual parallelism that is now available on stock hardware. Because it is easier to program with `fthread` than with `pthread`, we strongly recommend to use the former as much as possible and leave the former for specially demanding applications. Both libraries are described in this chapter.

16.1 Thread Common Functions

Bigloo implements SRFI-18 (Multithreading support). This SRFI is available at `http://srfi.schemers.org/srfi-18/srfi-18.html`. As Bigloo's threads are objects (see Section Chapter 9 [Object System], page 117), the SRFI-18's thread specific functions can be used with either the `pthread` or the `fthread` library.

This section describes the functions that are available independently of the multi-threading library.

16.1.1 Thread API

Bigloo uses a set of *primitive* functions and methods to create, run and handle thread. For the sake of standardization the name and semantic of SRFI-18 has been used. This section presents only the mandatory functions to program with threads in Bigloo.

The most important difference with SRFI-18, is the missing of the function `make-thread`, which is not available for all libraries, as it can be hard to predict the type of thread which will be created if several thread libraries are used simultaneously. As threads are regular Bigloo objects, they can be created using the `instantiate` syntax. See the Section 16.2 [Fair Threads], page 185, and Section 16.3 [Posix Threads], page 193, specific sections for more details about thread creation and examples.

The examples given in this section use a *generic* syntax with `instantiate::thread`, to run the examples, you will have to put them in a function in a module (see Section Chapter 2 [Modules], page 7, and import one of the libraries using `library` module declaration.

`current-thread` [SRFI-18 function]
: Returns the current thread.

`thread?` *obj* [SRFI-18 function]
: Returns `#t` if *obj* is a thread, otherwise returns `#f`.

`thread-name` *thread* [SRFI-18 function]
: Returns the name of the *thread*.

`thread-specific` *thread* [SRFI-18 function]
`thread-specific-set!` *thread obj* [SRFI-18 function]
: Returns and sets value in the specific field of the *thread*. If no value has been set, `thread-specific` returns an unspecified value.

```
(let ((t (instantiate::thread
```

```
            (body (lambda ()
                     (print (thread-specific (current-thread))))))))
   (thread-specific-set! t 'foo)
   (thread-start! t)) ⊣ foo
```

`thread-cleanup` *thread* [Bigloo function]
`thread-cleanup-set!` *thread fun* [Bigloo function]

Associates a cleanup function to a thread. The cleanup function is called with the thread itself. The cleanup function is executed in a context where `current-thread` is the thread owning the cleanup function.

```
(let ((t (instantiate::thread (body (lambda () 'done) 'foo))))
   (thread-cleanup-set! t (lambda (v) (print (thread-name (current-thread))
     ", exit value: " v)))
   (thread-start! t)) ⊣ foo, exit value: done
```

`thread-parameter` *ident* [Bigloo function]
`thread-parameter-set!` *ident value* [Bigloo function]

Returns the value of the parameter *ident* in the current thread. If no value is bound to this parameter, `#f` is returned.

A thread parameter is implemented by a chunk of memory specific to each thread. All threads are created with an empty set of parameters.

The next functions have different behaviors depending in the library used, more details will be given in the specific sections below.

`thread-start!` *thread* [*args*] [SRFI-18 function]
`thread-start-joinable!` *thread* [Bigloo function]
`thread-join!` *thread* [*timeout*] [SRFI-18 function]
`thread-terminate!` *thread* [SRFI-18 function]
`thread-yield!` [SRFI-18 function]
`thread-sleep!` *timeout* [SRFI-18 function]

16.1.2 Mutexes

Thread locking mechanism is common to Fair Threads and Posix Threads.

`mutex?` *obj* [SRFI-18 function]
`make-mutex` [*name*] [SRFI-18 function]
`make-spinlock` [*name*] [SRFI-18 function]
`mutex-name` *mutex* [SRFI-18 function]
`mutex-specific` *mutex* [SRFI-18 function]
`mutex-specific-set!` *mutex obj* [SRFI-18 function]
`mutex-state` *mutex* [SRFI-18 function]
`mutex-lock!` *mutex* [*timeout* [*thread*]] [SRFI-18 function, deprecated]
`mutex-unlock!` *mutex* [SRFI-18 function, deprecated]

The function `make-spinlock` creates a spin lock on architectures on support it, otherwise it creates a regular mutex as if `make-mutex` was called. The support for spin lock can be checked with:

```
bigloo-config 'have-spinlock
```

```
(let ((m (make-mutex)))
```

```
   (thread-start!
    (instantiate::thread
       (body (lambda ()
                (let loop ()
                   (if (mutex-lock! m 0)
                       (begin
                          (display "locked")
                          (mutex-unlock! m))
                       (begin
                          (thread-yield!)
                          (loop)))))))))
  ⊣ locked

(let ((res '()))
   (define (mutex-lock-recursively! mutex)
      (if (eq? (mutex-state mutex) (current-thread))
          (let ((n (mutex-specific mutex)))
             (mutex-specific-set! mutex (+ n 1)))
          (begin
             (mutex-lock! mutex)
             (mutex-specific-set! mutex 0))))
   (define (mutex-unlock-recursively! mutex)
      (let ((n (mutex-specific mutex)))
         (if (= n 0)
             (mutex-unlock! mutex)
             (mutex-specific-set! mutex (- n 1)))))
   (thread-start!
    (instantiate::thread
       (body (lambda ()
                (let ((m (make-mutex)))
                   (mutex-lock-recursively! m)
                   (mutex-lock-recursively! m)
                   (mutex-lock-recursively! m)
                   (set! res (cons (mutex-specific m) res))
                   (mutex-unlock-recursively! m)
                   (mutex-unlock-recursively! m)
                   (mutex-unlock-recursively! m)
                   (set! res (cons (mutex-specific m) res)))))))
   res)
  ⇒ (0 2)
```

`synchronize` *mutex exp1 exp2 ...* [Bigloo form]

The function *synchronize* evaluates the expressions *exp1*, *exp2*, etc. The mutex *mutex* is acquired and released before *exp1* gets evaluated. Its value is the value of the evaluated expression. The form `synchronize` ensures that however the form returns, the mutex *mutex* is always unlocked.

```
(synchronize mutex
   (print "before read...")
   (read p))
```

`with-lock` *mutex thunk* [Bigloo function, deprecated]

The form `with-lock` is similar to `synchronize` into which it is expanded.

The function *with-lock* evaluates the body of the *thunk*. The mutex *mutex* is acquired and released before *thunk* gets invoked. The function *with-lock* might be implemented as:

```
(define (with-lock mutex thunk)
```

```
(synchronize mutex
   (thunk)))
```

16.1.3 Condition Variables

`condition-variable?` *obj* [SRFI-18 function]
`make-condition-variable` [*name*] [SRFI-18 function]
`condition-variable-name` *cv* [SRFI-18 function]
`condition-variable-specific` *cv* [SRFI-18 function]
`condition-variable-specific-set!` *cv obj* [SRFI-18 function]
`condition-variable-wait!` *cv mutex* [*timeout*] [Bigloo function]
`condition-variable-signal!` *cv* [SRFI-18 function]
`condition-variable-broadcast!` *cv* [SRFI-18 function]

```
(let ((res 0))
   (define (make-semaphore n)
      (vector n (make-mutex) (make-condition-variable)))
   (define (semaphore-wait! sema)
      (mutex-lock! (vector-ref sema 1))
      (let ((n (vector-ref sema 0)))
         (if (> n 0)
             (begin
                (vector-set! sema 0 (- n 1))
                (mutex-unlock! (vector-ref sema 1)))
             (begin
                (condition-variable-wait! (vector-ref sema 2) (vector-ref sema 1))
                (mutex-unlock! (vector-ref sema 1))
                (semaphore-wait! sema)))))
   (define (semaphore-signal-by! sema increment)
      (mutex-lock! (vector-ref sema 1))
      (let ((n (+ (vector-ref sema 0) increment)))
         (vector-set! sema 0 n)
         (if (> n 0)
             (condition-variable-broadcast! (vector-ref sema 2)))
         (mutex-unlock! (vector-ref sema 1))))
   (let ((sema (make-semaphore 10)))
      (let ((t1 (thread-start!
                 (instantiate::thread
                    (body (lambda ()
                             (semaphore-wait! sema)
                             (set! res (current-time)))))))
            (t2 (thread-start!
                 (instantiate::thread
                    (body (lambda ()
                             (let loop ((n 10))
                                (if (> n 0)
                                    (begin
                                       (semaphore-signal-by! sema 1)
                                       (thread-yield!)
                                       (loop (- n 1))))))))))
         (scheduler-start!)
         res)))
⇒ 2
```

16.2 Threads

Bigloo supports fair threads (see Section Section 16.2.2.1 [Thread], page 186), a specification of cooperative threads. In this framework a thread must explicitly or implicitly *yield* the processor to the scheduler (see Section Section 16.2.2.2 [Scheduler], page 191). Explicit cooperation is achieved by library functions such as `thread-yield!` or `thread-sleep!`. The scheduler **does not preempt a running thread** to allocate the processor to another waiting thread. Fair threads have two drawbacks over preemptive threads:

- Cooperative threads are not able to benefit of multi-processors platforms.
- Single threads programs must be adapted in order to be run concurrently.

On the other hand, Fair threads have advantages that make them suitable for a high level programming language such as Scheme:

- Fair threads have a strong and well defined semantic. Multi threaded programs using Fair threads are *deterministic* thus programs that deploy Fair threads are *predictable*.
- Fair threads are easier to program with because they hide most the of the concurrent programming pitfalls. In particular, since Fair threads enforce a strong synchronization, there is no need to deploy techniques such as *mutex*, *semaphore* or *condition variables*.

This whole chapter has been written in collaboration with **F. Boussinot**. It uses materials on Fair threads that can be found at `http://www-sop.inria.fr/indes/rp/FairThreads/html/FairThreads.html`.

16.2.1 Introduction to Fair Threads

Fair threads are cooperative threads run by a fair scheduler which gives them equal access to the processor. Fair threads can communicate using broadcast events. Their semantics does not depends on the executing platform. Fine control over fair threads execution is possible allowing the programming of specific user-defined scheduling strategies.

Contrary to standard sequential programming where the processor executes a single program, in concurrent programming the processor is a shared resource which is dispatched to several programs. The term *concurrent* is appropriate because programs can be seen as concurrently competing to gain access to the processor, in order to execute.

Threads are a basic means for concurrent programming, and are widely used in operating systems. At language level, threads offer a way to structure programs by decomposing systems in several concurrent components; in this respect, threads are useful for modularity.

However, threads are generally considered as low-level primitives leading to over-complex programming. Moreover, threads generally have loose semantics, in particular depending on the underlying executing platform; to give them a precise semantics is a difficult task, and this is a clearly identified problem to get portable code.

Bigloo proposes a new framework with clear and simple semantics, and with an efficient implementation. In it, threads are called *fair*; basically a fair thread is a cooperative thread executed in a context in which all threads always have equal access to the processor. Fair threads have a deterministic semantics, relying on previous work belonging to the so-called reactive approach.

16.2.2 Fair Threads Api

The Fair Thread library relies on the Posix Thread one, but you don't need to import the `pthread` library, as it is done automatically when importing the `fthread` one.

The functions listed in Section 16.1 [Thread Common Functions], page 181, can be used to manipulates the Fair Thread, but `thread-start-joinable!`, as a fair thread can always join any other fair thread in the same scheduler.

16.2.2.1 Thread

`instantiate::fthread` (*body* `thunk`) [(*name* `name`)] [Bigloo syntax]

Returns a new thread which is not started yet. The body of the thread is the body of the procedure *thunk*. The optional argument *name* can be use to identify the thread. It can be any Bigloo value.

```
(instantiate::fthread (body (lambda () (print 1) (thread-yield!)
(print 2)))
                      (name 'my-thread))
```

The former thread-start function can be rewritten as follow:

```
(define (make-thread body . name)
  (if (pair? name)
      (instantiate::fthread (body body) (name (car name)))
      (instantiate::fthread (body body))))
```

`thread-start!` *thread* [*scheduler*] [SRFI-18 function]

Runs a thread created with `make-thread`. If *scheduler* is provided, the *thread* is started in this particular scheduler. Otherwise, it is started in the current scheduler (see Section Section 16.2.2.2 [Scheduler], page 191). Threads are started at the beginning of *reactions* (see Section Section 16.2.2.2 [Scheduler], page 191).

`thread-yield!` [SRFI-18 function]

The current thread *cooperates*. That is, it is suspended for the *reaction* and the scheduler selects a new thread to be resumed. The scheduler resumes the next available thread. If there is only one thread started in the scheduler, the same thread is resumed. A *reaction* corresponds to the invocation of a `scheduler-react!` call (see Section Section 16.2.2.2 [Scheduler], page 191).

`thread-sleep!` *timeout* [SRFI-18 function]

The current thread *cooperates* during *exactly timeout reactions* (see Section 16.2.2.2 [Scheduler], page 191). It is suspended and the scheduler selects a new thread to be resumed. If there is only one thread started in the scheduler, the same thread will be resumed.

```
(let ((t1 (instantiate::fthread
             (body (lambda () (thread-sleep! 2) (display 'foo)))))
      (t2 (instantiate::fthread
             (body (lambda () (let loop ((n 1))
                                 (display n)
                                 (thread-yield!)
                                 (if (< n 5)
                                 (loop (+ n 1)))))))))
  (thread-start! t1)
  (thread-start! t2)
  (scheduler-start!)) ⊣ 12foo34
```

`thread-terminate!` *thread* [SRFI-18 function]

Terminates *thread* at the end of the current reaction.

`thread-join!` *thread* [*timeout* [*timeout-val*]] [SRFI-18 function]

The current thread waits until *thread* terminates or until *timeout* is reached (when supplied). If the *timeout* is reached, `thread-join!` returns *timeout-val*. If *thread* terminates, `thread-join!` returns the end-result of the *thread* or the end-exception if that thread terminates abnormally.

If several threads wait for the termination of the same thread, they are all notified of the termination during the current reaction.

```
(let* ((t1 (thread-start!
            (instantiate::fthread
               (body (lambda () (thread-sleep! 3) 'foo)))))
       (t2 (thread-start!
            (instantiate::fthread
               (body (lambda () (print "t1: " (thread-join! t1 1)))))))
       (t3 (thread-start!
            (instantiate::fthread
               (body (lambda () (print "t2: " (thread-join! t1 2 'bar)))))))
       (t3 (thread-start!
            (instantiate::fthread
               (body (lambda () (print "t3: " (thread-join! t1)))))))
       (t4 (thread-start!
            (instantiate::fthread
               (body (lambda () (print "t4: " (thread-join! t1))))))))
  (scheduler-start!))
 ⊣ t1: #|%uncaught-exception [reason: (exception . join-timeout)]|
   t2: bar
   t3: foo
   t4: foo
```

`thread-join!` can be used to wait for a Posix Thread termination. The `pthread` object must be started with `thread-start-joinable!`.

`thread-suspend!` *thread* [Bigloo function]

`thread-resume!` *thread* [Bigloo function]

Suspends/resumes the *thread* at the end of reaction. While suspended a thread is not eligible to get the processor by the scheduler.

`thread-await!` *signal* [*timeout*] [Bigloo function]

Blocks the thread until *signal* has been broadcast or until *timeout* has elapsed. The function `thread-await!` returns the value associated with the previous emissions of the signal that took place during the reaction.

```
(let ((t1 (thread-start! (instantiate::fthread
                           (body (lambda ()
                                    (display (thread-await! 'foo))
                                    (display (thread-await! 'bar)))))))
      (t2 (thread-start! (instantiate::fthread
                           (body (lambda ()
                                    (broadcast! 'foo 'val1-foo)
                                    (broadcast! 'foo 'val2-foo))))))
      (t3 (thread-start! (instantiate::fthread
                           (body (lambda ()
                                    (thread-sleep! 2)
```

```
                                        (broadcast! 'bar 'val-bar)))))))
   (let loop ((n 1))
      (display n)
      (scheduler-react! (default-scheduler))
      (loop (+ n 1))))
  ⊣ 1val2-foo23val-bar456...
```

The function `thread-await!` cannot be used to intercept all the signals broadcast during a reaction. This is illustrated by the following example where obviously `thread-await!` cannot intercept the emission of the signal:

```
(thread-start! (instantiate::fthread (body (lambda ()
                                              (thread-await! 'foo)
                                              (broadcast! 'foo 1)))))
(thread-start! (instantiate::fthread (body (lambda ()
                                              (broadcast! 'foo 2)))))
```

`thread-get-values!` *signal* [Bigloo function]

Terminates the instant for the thread (as `thread-yield!`) and returns, hence at the next instant, all the values associated with broadcast *signal* (see Section Section 16.2.2.3 [Signal], page 192) during the previous scheduler reaction (see Section Section 16.2.2.2 [Scheduler], page 191).

Example:

```
(thread-start! (instantiate::fthread
                  (body (lambda ()
                           (for-each print (thread-get-values! 'foo))))))
(thread-start! (instantiate::fthread
                  (body (lambda ()
                           (broadcast! 'foo 1)
                           (broadcast! 'foo 'foo)
                           (broadcast! 'foo "blabla")))))
   ⊣ 1
     foo
     blabla
```

Example:

```
(let ((t1 (thread-start!
           (instantiate::fthread
              (body (lambda ()
                       (for-each print (thread-get-values! 'foo))))
              (name 't1))))
      (t2 (thread-start!
           (instantiate::fthread
              (body (lambda ()
                       (broadcast! 'foo (current-thread))
                       (thread-yield!)
                       ;; this second broadcast won't be intercepted
                       ;; because it occurs during the next reaction
                       (broadcast! 'foo (current-thread))))
              (name 't2))))
      (t3 (thread-start!
           (instantiate::fthread
              (body (lambda ()
                       (broadcast! 'foo (current-thread))
                       (broadcast! 'foo (current-thread))))
              (name 't3)))))
  (scheduler-start!))
  ⊣ #<thread:t2>
```

```
#<thread:t3>
#<thread:t3>
```

`thread-await-values!` *signal* [*timeout*] [Bigloo function]

This blocks the current thread until *signal* has been broadcast. It then returns, at the next instant, all the values associated with all the broadcasts that took place during the instant. It can be defined as:

```
(define (thread-await-values! signal . tmt)
   (apply thread-await! signal tmt)
   (thread-get-values signal))
```

`thread-await*!` *signals* [*timeout*] [Bigloo function]

Wait for one of a list of signals. The function `thread-await*!` can be compared to the Unix `select` function. The argument *signals* is a list of signal identifier. The function `thread-await*!` blocks the current thread until one of the signal in the list *signals* is broadcast or until the optional numerical argument *timeout* is elapsed. If the thread unblocks because the timeout is elapsed, `thread-await*!` returns `#f`. Otherwise it returns two values that have to be collected with `multiple-value-bind` (see Section 5.1.13 [Control Features], page 50). The first one is the value of the broadcast signal. The second one is the broadcast signal.

Example:

```
(let ((res #f))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (let ((sig* (list 'foo 'bar)))
                   (multiple-value-bind (val1 sig1)
                      (thread-await*! sig*)
                      (multiple-value-bind (val2 sig2)
                         (thread-await*! sig*)
                         (thread-yield!)
                         (multiple-value-bind (val3 sig3)
                            (thread-await*! sig*)
                            (set! res (list sig1 sig2 sig3)))))))))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-sleep! 2)
                (broadcast! 'foo 1)))))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-sleep! 3)
                (broadcast! 'bar 2)))))
   (scheduler-start!)
   res)
  ⇒ '(foo foo bar)
```

A second example using timeouts:

```
(let ((res #f))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (let ((sig* (list 'foo 'bar)))
                   (multiple-value-bind (val1 sig1)
```

```
                     (thread-await*! sig* 1)
                     (thread-yield!)
                     (multiple-value-bind (val2 sig2)
                        (thread-await*! sig* 1)
                        (thread-yield!)
                        (multiple-value-bind (val3 sig3)
                           (thread-await*! sig* 2)
                           (set! res (list sig1 sig2 sig3))))))))))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-sleep! 2)
                (broadcast! 'foo 1)))))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-sleep! 3)
                (broadcast! 'bar 2)))))
   (scheduler-start!)
   res)
  ⇒ '(#f foo bar)
```

`thread-get-values*!` *signals* [Bigloo function]

Terminates the instant for the thread (as `thread-yield!`) and returns, hence at the next instant, all the values associated with all broadcast *signals* (see Section Section 16.2.2.3 [Signal], page 192) during the previous scheduler reaction (see Section Section 16.2.2.2 [Scheduler], page 191). The function `thread-get-values*!` returns an *alist* made of the scanned signal and their values. That is the length of the returns list is the length of the list *signals*. If a signal of the list *signals* has not been broadcast, its associated entry the list returned by `thread-get-values*!` has an empty `cdr`.

Example:

```
(let ((s1 'foo)
      (s2 'bar)
      (s3 'gee)
      (res #f))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-sleep! 2)
                (broadcast! 'foo (current-time))
                (broadcast! 'bar 0)))))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-await*! (list s1 s2 s3))
                (set! res (thread-get-values*! (list s1 s2 s3)))))))
   (thread-start!
    (instantiate::fthread
       (body (lambda ()
                (thread-sleep! 2)
                (broadcast! 'bar (current-time))))))
   (scheduler-start!)
   res)
  ⇒ ((foo 3) (bar 3 0) (gee))
```

Used with asynchronous signal, the functions `thread-await*!` and `thread-get-values*!` can be used to read concurrently, in a non blocking way, several files.

`thread-await-values*!` *signals* [*timeout*] [Bigloo function]
This blocks the current thread until at least one of *signals* has been broadcast. It then returns, at the next instant, all the values associated with all the broadcasts that took place during the instant. It can be defined as:

```
(define (thread-await-values*! signal . tmt)
   (apply thread-await*! signal tmt)
   (thread-get-values*! signal))
```

16.2.2.2 Scheduler

`make-scheduler` [*strict-order?*] [*envs*] [Bigloo function]
Creates a new scheduler. The optional boolean argument `strict-order?` is used to ask the scheduler to always schedule the threads in the same order, it defaults to `#f`. The optional arguments *envs* are *fair thread environments* which will be defined in forthcoming Bigloo releases.

`scheduler-strict-order?` [Bigloo function]
`scheduler-strict-order?-set!` *bool* [Bigloo function]
Gets or sets the strict scheduling policy of the scheduler. If set, the threads will always be scheduled in the same order, until their termination. By default, it is set to false, which improve performances when there is a lot of thread to schedule.

`scheduler?` *obj* [Bigloo function]
Returns `#t` if *obj* is a scheduler. Otherwise returns `#f`.

`scheduler?` *obj* [Bigloo function]
Returns `#t` if *obj* is a scheduler. Otherwise returns `#f`.

`current-scheduler` [Bigloo function]
Returns the current scheduler. The current scheduler is the scheduler which currently schedules the current thread. This value is not mutable, as it is set during the call to `thread-start!`.

`default-scheduler` [*scheduler*] [Bigloo function]
Sets or gets the default scheduler. The default scheduler is the scheduler that will be used in the calls to `scheduler-react!`, `scheduler-start!` or `thread-start!` if not specified. It always exists a default scheduler. That is, it is optional for an application to create a scheduler.

`scheduler-react!` [*scheduler*] [Bigloo function]
Executes all the threads started (see `thread-start!`, Section Section 16.2.2.1 [Thread], page 186) in the scheduler until all the threads are blocked. A thread is blocked if the has explicitly yield the processor (`thread-yield!` and `thread-sleep!`) or because it is waiting a signal (`thread-await!`). A thread can be selected several times during the same reaction. The function `scheduler-react!` returns a symbol denoting the *state* of the scheduler. The possible states are:

- `ready` The Scheduler is ready to execute some threads.

- `done` All the threads started in the scheduler have terminated.
- `await` All the threads started in the scheduler are waiting for a signal.

An invocation of `scheduler-react!` is called a *reaction*.

`scheduler-start!` [*arg* [*scheduler*]] [Bigloo function]
Executes `scheduler-react!` as long as the scheduler is not done. If the optional argument *scheduler* is not provided, `scheduler-start!` uses the current scheduler (see `current-scheduler`). The optional *arg* can either be:

- An integer standing for the number of times `scheduler-react!` must be called.
- A procedure *f* of one argument. The procedure *f* is invoked after each reaction. It is passed a value *i* which is the iteration number of the scheduler. The reactions of the scheduler continue while *f* returns `#f`.

```
(let* ((s (make-scheduler))
       (t (instantiate::fthread
             (body (lambda ()
                      (let loop ((n 0))
                         (display n)
                         (thread-yield!)
                         (loop (+ 1 n))))))))
   (scheduler-start! 10 s))
  ⊣ 0123456789

(let* ((s (make-scheduler))
       (t (instantiate::fthread
             (body (lambda ()
                      (let loop ((n 0))
                         (display n)
                         (thread-yield!)
                         (loop (+ 1 n))))))))
   (scheduler-start! (lambda (i) (read-char)) s))
  ⊣ 0123456789
```

`scheduler-terminate!` [*scheduler*] [Bigloo function]
Terminates all the threads in *scheduler*.

`scheduler-instant` [*scheduler*] [Bigloo function]
Returns the current reaction number of *scheduler*. The reaction number is the number of times `scheduler-react!` has been invoked passing *scheduler* as argument.

16.2.2.3 Signal

`broadcast!` *signal* [*val*] [Bigloo function]
Broadcasts *signal* to all threads started in *scheduler* *immediately*, that is during the reaction. This function can only be called from within a running thread. If the optional argument *val* is omitted, the *signal* is broadcast with an unspecified value.

```
(thread-start! (instantiate::fthread
                  (body (lambda ()
                           (thread-await! 'foo)
                           (print (scheduler-instant (current-scheduler)))))))
(thread-start! (instantiate::fthread
                  (body (lambda ()
                           (broadcast! 'foo)))))
```

```
(scheduler-start!)
  ⊣ 1
```

`scheduler-broadcast!` *scheduler signal* [*val*] [Bigloo function]
: At the next react broadcasts *signal* to all threads started in *scheduler*. This is used to impact running threads from outside any threads. If the optional argument *val* is omitted, the *signal* is broadcast with an unspecified value.

`make-asynchronous-signal` *proc* [Bigloo function]
: This function invokes in the background, the procedure *proc*. This function takes one parameter which is the signal that is broadcast when the invocation returns. When the host operating system supports parallel executions, the invocation of *proc* is executed in parallel with the waiting thread.

  Asynchronous signals can be used to implement non blocking system operations, such as input/output. Here is an example that illustrates how to implement concurrent programs that behaves similarly with Fair Threads and Posix Threads.

```
(define-expander read
   (lambda (x e)
      (cond-expand
 (fthread
  (thread-await!
   (make-aynchronous-signal
    (lambda (s)
       (read ,@(map (lambda (x) (e x e)) (cdr x)))))))
 (else
  `(read ,@(map (lambda (x) (e x e)) (cdr x)))))))
```

16.2.3 SRFI-18

This section presents the functions that are not necessary to Bigloo but supported for compliance with SRFI-18, provided by the Fair Thread library.

`current-time` [*scheduler*] [SRFI-18 function]
: Returns the reaction number of *scheduler*.

`time?` *obj* [SRFI-18 function]
`time->seconds` *obj* [SRFI-18 function]

`join-timeout-exception?` *obj* [SRFI-18 function]
`abandoned-mutex-exception?` *obj* [SRFI-18 function]
`terminated-thread-exception?` *obj* [SRFI-18 function]
`uncaught-exception?` *obj* [SRFI-18 function]
`uncaught-exception-reason` *exc* [SRFI-18 function]

16.3 Posix Threads

This section describes two Posix-Like multi-threading Bigloo libraries. The two libraries, `pthread`, and `srfi-18` are all the same but the `mutex-state` function that returns different results. Because of these differences that might seem thin at first glance, the `pthread` library is significantly faster than the `srfi-18` library. For that reason, it is recommended to use the `pthread` library instead of the `srfi-18` library that is mostly supported for backward compatibility.

As much as possible, the names exported by this library are compatible with the Fair Threads library (see Section Section 16.2 [Fair Threads], page 185).

16.3.1 Using Posix Threads

The Bigloo modules initialization model does not permit to create threads before the main function is started. In other words, it is unsafe to use the Posix Threads API at the top level of modules. On some particular applications this might work correctly. On other it could produce an error message stating the threads cannot be created or started before the pthread library is initialized.

16.3.2 Threads

`instantiate::pthread` (*body* `thunk`) [(*name* `name`)] [Bigloo syntax]
`make-thread` *thunk* [*name*] [SRFI-18 function]

Returns a new thread which is not started yet. The body of the thread is the body of the procedure *thunk*. The optional argument *name* can be use to identify the thread. It can be any Bigloo value.

Warning: the `make-thread` function is **deprecated**, but still provided for a backward compatibility with previous release of Bigloo. The use of this function is *highly* discouraged, in favor of the `instantiate::pthread` form.

```
(module example
   (library pthread)
   (main main))

(define (main argv)
   (make-thread
    (lambda ()
       (print 1)
       (thread-yield!)
       (print 2))
    'my-thread))
```

`thread-start!` *thread* [SRFI-18 function]
`thread-start-joinable!` *thread* [SRFI-18 function]

Runs a thread created with `instantiate::pthread`. By default, threads are detached, and thus, they cannot be joined.

`thread-yield!` [SRFI-18 function]

The current thread *cooperates*.

`thread-sleep!` *timeout* [SRFI-18 function]

The current thread *sleeps* for a certain period. It is suspended and the scheduler is free to select a new thread to be resumed. If there is only one thread started in the scheduler, the same thread will be resumed. The time of *timeout* is used to determine the time the thread must sleep.

Here are the possible types for *timeout*.

- `date`: the thread sleeps at least until the date *timeout*.
- `real`: the thread sleeps at least *timeout* seconds.
- `fixum`, `elong`, `llong`: the thread sleeps at least *timeout* milli-seconds.

`thread-terminate!` *thread* [SRFI-18 function]

Terminates *thread* as soon as possible.

`thread-join!` *thread* [*timeout*] [SRFI-18 function]

The current thread waits until the *thread* terminates. If *thread* terminates, `thread-join!` returns the end-result of the *thread* or the end-exception if that thread terminates abnormally.

It is possible to wait for the termination of the a thread if and only if it has been started with `thread-start-joinable!`. In particular, threads started with `thread-start!` cannot be joined.

The optional argument *timeout*, forces to wait at for *timeout* milli-seconds for the thread to terminate. Note that not all systems support this facility. When supported, the `cond-expand` (see see Chapter 30 [SRFIs], page 269) `pthread-timedjoin` is defined. When the timeout expires some systems, raise an error. Other systems abort silently.

`terminated-thread-exception?` *obj* [SRFI-18 function]
`uncaught-exception?` *obj* [SRFI-18 function]
`uncaught-exception-reason` *exc* [SRFI-18 function]

16.3.3 Mutexes

Thread locking mechanism is common to Fair Threads and Posix Threads (see Section 16.1 [Thread Common Functions], page 181).

`mutex-state` *mutex* [SRFI-18 function]

Returns the symbol `locked` when the mutex is locked by a thread. Otherwise, it returns the symbol `unlocked`.

16.3.4 Condition Variables

Posix thread condition variables follows the common thread API (see Section 16.1 [Thread Common Functions], page 181).

```
(module example
  (library pthread)
  (main main))

(define (main argv)
   (let ((res #f)
 (lock (make-mutex))
 (cv (make-condition-variable)))
      (thread-join!
       (thread-start-joinable!
(instantiate::pthread
   (body (lambda ()
    (mutex-lock! lock)
    (thread-start!
     (instantiate::pthread
(body (lambda ()
 (mutex-lock! lock)
 (condition-variable-signal! cv)
 (mutex-unlock! lock)))))
   (condition-variable-wait! cv lock)
```

```
         (set! res 23)
         (mutex-unlock! lock))))))
           res))
```

16.3.5 Semaphores

`open-semaphore` *name* [*create* `#t`] [*excl* `#t`] [*read* `#t`] [*write* `#t`] [*value* `1`] [Bigloo function]

`close-semaphore` *sem* [Bigloo function]

`delete-semaphore` *name* [Bigloo function]

Opens, closes, and deletes a named semaphore.

`semaphore?` *obj* [Bigloo function]

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

`semaphore-post` *sem* [Bigloo function]

`semaphore-wait` *sem* [Bigloo function]

`semaphore-trywait` *sem* [Bigloo function]

Wait and post for a semaphore.

`semaphore-value` *sem* [Bigloo function]

Returns the current value of the semaphoe

16.3.6 SRFI-18

`mutex-state` *mutex* [SRFI-18 function]

Returns information about the state of the mutex. The possible results are:

- thread T: the mutex is in the locked/owned state and thread T is the owner of the mutex
- symbol `not-owned`: the mutex is in the locked/not-owned state
- symbol `abandoned`: the mutex is in the unlocked/abandoned state
- symbol `not-abandoned`: the mutex is in the unlocked/not-abandoned state

Examples:

```
(mutex-state (make-mutex))
   ⇒ not-abandoned

(define (thread-alive? thread)
   (let ((mutex (make-mutex)))
      (mutex-lock! mutex #f thread)
      (let ((state (mutex-state mutex)))
         (mutex-unlock! mutex) ; avoid space leak
         (eq? state thread))))
```

16.4 Mixing Thread APIs

The Section 16.2 [Fair Threads], page 185, library is "Posix Threads" safe, which means it is possible to use at the same time both libraries. In other words, it is possible to embed one fair scheduler into a Posix thread.

Here is a little example with two schedulers started into two different Posix threads, each schedulers running two fair threads.

```
(module mix_threads
   (library fthread pthread)
   (main main))

(define *f1* 0)
(define *f2* 0)

(define (main args)
   (let ((s1 (make-scheduler #t))
 (s2 (make-scheduler #t))

 (f1a (instantiate::fthread
 (body (lambda ()
  (let loop ()
     (print "f1a: " *f1* " " (current-thread))
     (set! *f1* (+ 1 *f1*))
     (thread-yield!)
     (loop))))))

 (f1b (instantiate::fthread
 (body (lambda ()
  (let loop ()
     (print "f1b: " *f1* " " (current-thread))
     (set! *f1* (+ 1 *f1*))
     (thread-yield!)
     (loop))))))

 (f2a (instantiate::fthread
 (body (lambda ()
  (let loop ()
     (print "f2a: " *f2* " " (current-thread))
     (set! *f2* (+ 1 *f2*))
     (thread-yield!)
     (loop))))))

 (f2b (instantiate::fthread
 (body (lambda ()
  (let loop ()
     (print "f2b: " *f2* " " (current-thread))
     (set! *f2* (+ 1 *f2*))
     (thread-yield!)
     (loop)))))))

      (let* ((p1 (instantiate::pthread
   (body (lambda ()
    ;; Sets the thread's specific scheduler
    (default-scheduler s1)
    (scheduler-start! 5)))))

    (p2 (instantiate::pthread
   (body (lambda ()
    ;; Sets the thread's specific scheduler
    (default-scheduler s2)
    ;; One reaction for s2
    (scheduler-react!)
    ;; Starts s1
```

```
   (thread-start-joinable! p1)
   ;; Do three reactions
   (scheduler-start! 3)
   ;; Waits for p1/s1 termination
   (thread-join! p1)
   ;; The final reaction
   (scheduler-react!))))))

(thread-start! f1a s1)
(thread-start! f1b s1)
(thread-start! f2a s2)
(thread-start! f2b s2)

(thread-join! (thread-start-joinable! p2)))))
```

17 Database

Bigloo supports database programming. The current version proposes a SQLite binding.

17.1 SQLite

The Bigloo's C back-end supports SQL queries. It relies on the SQLite library (`http://www.sqlite.org/`). The SQLite binding is accessible to Bigloo via the `sqlite` library. Here is an example of module that uses this library.

```
(module example1
  (library sqlite))

(let ((db (instantiate::sqlite)))
  ...)
```

`sqlite` [Bigloo Sqlite class]

```
(class sqlite
   (path::bstring read-only (default ":memory:")))
```

The instances of the class `sqlite` hold SQLite databases. A database may be permanently stored on a disk or loaded in memory. The class attribute `path` is the location on the disk where the database is stored. The special path `:memory:` denotes in-memory databases. When an instance is created a SQLite database is *opened*.

Example:

```
(define db1 (instantiate::sqlite (path "/tmp/foo.db")))
(define db2 (instantiate::sqlite))
```

Binds the global variable *db1* to a database that is stored on the file system at location `/tmp/foo.db`. This example also binds the global variable *db2* to an in-memory SQLite database.

`sqlite-close` *sqlite* [Bigloo Sqlite function]

This function closes a database previously opened by creating an instance of the class `sqlite`.

Example:

```
(let ((db (instantiate::sqlite)))
  (sqlite-exec db "CREATE TABLE table1 (x INTEGER, y INTEGER)")
  (sqlite-exec db "INSERT INTO table1 VALUES( ~a, ~a )" 1 4)
  (sqlite-close db))
```

`sqlite-format` *string arg ...* [Bigloo Sqlite function]

Constructs a string of characters representing an SQLite commands. This function acts as `format` (see Section 5.2 [Input and Output], page 54). It is augmented with three additional escape sequence: `~q`, `~k`, and `~l`. The first one build a string of characters where the characters denoting SQL strings (i.e., the character `'`) is automatically escaped. The escape character `~k` introduces a list of SQL strings. The escape character `~l` introduces a SQL list.

Summary of all escape codes:

- `~a` The corresponding value is inserted into the string as if printed with display.
- `~s` The corresponding value is inserted into the string as if printed with write.

- `~%` A newline is inserted.
- `~~` A tilde `~` is inserted.
- `~q` An SQL escaped string.
- `~l` Introduces a list (comma separated).
- `~k` Introduces a list of SQL strings.

Examples:

```
(module example
   (library sqlite))

(sqlite-format "~a" "foo'bar") ⇒ "foo'bar"
(sqlite-format "~q" "foo'bar") ⇒ "'foo''bar'"
(sqlite-format "~a" '("foo'bar" "foo")) ⇒ "(foo'bar foo)"
(sqlite-format "~k" '("foo'bar" "foo")) ⇒ "'foo''bar','foo'"
(sqlite-format "~l" '("foo'bar" "foo")) ⇒ "foo'bar,foo"
```

`sqlite-exec` *sqlite string arg ...* [Bigloo Sqlite function]

The function `sqlite-exec` *executes* an SQLite command. The command is the built by implicitly invoking `sqlite-format` on *string* and the optional *arg* arguments. This function returns a single element, the first one returned by the SQL engine.

Example:

```
(module example
   (library sqlite))

(define *db* (instantiate::sqlite))

(sqlite-exec *db* "CREATE TABLE foo (x INTEGER, y INTEGER)")
(for-each (lambda (x)
(sqlite-exec *db*  "INSERT INTO foo VALUES(~A, ~A)" x (* x x)))
     (iota 10))
(sqlite-exec *db* "SELECT * FROM foo")
   ⇒ 9
```

`sqlite-eval` *sqlite procedure string arg ...* [Bigloo Sqlite function]

The function `sqlite-eval` invokes a SQLite command built by implicitly invoking `sqlite-format` on *string* and the optional *arg* arguments. The result of the function is built by applying *procedure* to the first value returned by the SQLite call.

Note: user callback (*procedure*) **must not** exit. That is they must not invoke a function create by `bind-exit`. Exiting from a callback will leave the database in a inconsistent state that prevent transactions to be rolled back.

`sqlite-map` *sqlite procedure string arg ...* [Bigloo Sqlite function]

The function `sqlite-map` invokes a SQLite command built by implicitly invoking `sqlite-format` on *string* and the optional *arg* arguments. The result is a list whose elements are built by applying *procedure* to all the values returned by the SQLite call.

Note: user callback (*procedure*) **must not** exit. That is they must not invoke a function create by `bind-exit`. Exiting from a callback will leave the database in a inconsistent state that prevent transactions to be rolled back. Example:

```
(module example
```

```
   (library sqlite))

(define *db* (instantiate::sqlite))

(sqlite-exec *db* "CREATE TABLE foo (x INTEGER, y INTEGER)")
(for-each (lambda (x)
(sqlite-exec *db*  "INSERT INTO foo VALUES(~A, ~A)" x (* x x)))
     (iota 10))
(sqlite-map *db*
  (lambda (s1 s2) (+ (string->integer s1) (string->integer s2)))
  "SELECT * FROM foo")
   ⇒ (0 2 6 12 20 30 42 56 72 90)
```

Example2:

```
(module example
   (library sqlite))

(define *db* (instantiate::sqlite))

(sqlite-exec *db* "CREATE TABLE foo (x INTEGER, y INTEGER)")
(for-each (lambda (x)
(sqlite-exec *db*  "INSERT INTO foo VALUES(~A, ~A)" x (* x x)))
     (iota 10))
(sqlite-map *db* vector "SELECT * FROM foo")
   ⇒ '(#("0" "0")
#("1" "1")
#("2" "4")
#("3" "9")
#("4" "16")
#("5" "25")
#("6" "36")
#("7" "49")
#("8" "64")
#("9" "81"))
```

`sqlite-name-of-tables` *`sqlite`* [Bigloo Sqlite function]

Returns the name of tables in the database. This list can also be obtained with

```
(sqlite-map db
   (lambda (x) x)
   "SELECT name FROM sqlite_master WHERE type='table'")
```

`sqlite-table-name-of-columns` *`sqlite table`* [Bigloo Sqlite function]

Returns the name of columns in the table.

`sqlite-last-insert-rowid` *`sqlite`* [Bigloo Sqlite function]

Returns the SQLite *rowid* of the last inserted row.

18 Multimedia

Bigloo provides various facilities for programming multimedia applications. It provides functions for parsing images and sounds and functions for controlling music players. All the functions, variables, and classes presented in the document are accessible via the `multimedia` library. Here is an example of module that uses this library:

```
;; Extract the thumbnail of a digital photography.
(module thumbnail
   (library multimedia)
   (main main))

(define (main argv)
   (when (and (pair? (cdr argv)) (file-exists? (cadr argv)))
      (let ((ex (jpeg-exif (cadr argv))))
          (when (exif? ex)
             (display (exif-thumbnail ex))))))
```

18.1 Photography

The multimedia library provides functions for accessing the metadata generated by digital camera.

`jpeg-exif` *file-name* [Bigloo Multimedia procedure]

The function `jpeg-exif` extracts the EXIF (`http://en.wikipedia.org/wiki/Exif`) metadata of a JPEG file as created by digital camera. The argument *file-name* is the name of the JPEG file. If the file contains an EXIF section it is returned as an instance of the `exif` class. Otherwise, this function returns `#f`.

`jpeg-exif-comment-set!` *file-name text* [Bigloo Multimedia procedure]

Set the comment of the EXIF metadata section of the file file-name to *text*.

`exif` [Bigloo Multimedia class]

```
(class exif
   (version (default #f))
   (jpeg-encoding (default #f))
   (jpeg-compress (default #f))
   (comment (default #f))
   (commentpos (default #f))
   (commentlen (default #f))
   (date (default #f))
   (make (default #f))
   (model (default #f))
   (orientation (default 'landscape))
   (width (default #f))
   (height (default #f))
   (ewidth (default #f))
   (eheight (default #f))
   (xresolution (default #f))
   (yresolution (default #f))
   (resolution-unit (default #f))
   (focal-length (default #f))
   (flash (default #f))
   (fnumber (default #f))
   (iso (default #f))
   (shutter-speed-value (default #f))
```

```
(exposure-time (default #f))
(exposure-bias-value (default #f))
(aperture (default #f))
(metering-mode (default #f))
(cdd-width (default #f))
(focal-plane-xres (default #f))
(focal-plane-units (default #f))
(thumbnail (default #f))
(thumbnail-path (default #f))
(thumbnail-offset (default #f))
(thumbnail-length (default #f)))
```

The instance of the `exif` class maps the EXIF metadata found in JPEG files into Bigloo objects. Since all fields are optional they are untyped.

`exif-date->date` [Bigloo Multimedia procedure]

Parses an exif date, i.e., a string of characters, and returns corresponding date. Raises an *&io-parse-error* if the string does not represents an exif date whose syntax is given by the following regular expression:

```
[0-9][0-9][0-9]:[0-9][0-9]:[0-9][0-9] :[0-9][0-9]:[0-9][0-9]:[0-9][0-9]
```

18.2 Music

The multimedia library provides an extensive set of functions for dealing with music. It provides functions for accessing the metadata of certain music file formats, it provides functions for controlling the volume of the hardware mixers and it provides functions for playing and controlling music playback.

18.2.1 Metadata and Playlist

`read-m3u` *input-port* [Bigloo Multimedia procedure]
`write-m3u` *list output-port* [Bigloo Multimedia procedure]

The function `read-m3u` reads a playlist expressed in the M3U format from *input-port* and returns a list of songs. The function `write-m3u` encode such a list encoded in the M3U format to an output port.

`file-musictag` *file-name* [Bigloo Multimedia procedure]
`mp3-musictag` *file-name* [Bigloo Multimedia procedure]
`ogg-musictag` *file-name* [Bigloo Multimedia procedure]
`flac-musictag` *file-name* [Bigloo Multimedia procedure]

These functions extract the metadata of a music file named *file-name*.

The function `mp3-musictag` returns the ID3 tag section if it exists. Otherwise, it returns `#f`. The function `ogg-musictag` and `flac-musictag` returns the vorbis comment if it exists.

`musictag` [Bigloo Multimedia class]

```
(abstract-class musictag
   (title::bstring read-only)
   (artist::bstring read-only)
   (orchestra::obj read-only (default #f))
   (interpret::obj read-only (default #f))
   (album::bstring read-only)
   (year::int read-only)
```

```
   (comment::bstring read-only)
   (genre::bstring read-only)
   (track::int (default -1)))
```

This class is used as the base class of music tag formats.

`id3::musictag` [Bigloo Multimedia class]

```
(class id3::musictag
   version::bstring
   (orchestra::obj read-only (default #f))
   (conductor::obj read-only (default #f))
   (recording read-only (default #f))
   (cd::obj (default #f)))
```

This class is used to reify the ID3 metadata used in the MP3 format.

`vorbis::musictag` [Bigloo Multimedia class]

```
(class vorbis::musictag)
```

This class is used to reify the Vorbis comments of OGG and Flac files.

18.2.2 Mixer

Bigloo proposes various functions and classes for controlling the audio volume of sound cards.

`mixer` [Bigloo Multimedia class]

```
(class mixer
   (devices::pair-nil (default '())))
```

The field *devices* is a list of available channels.

`mixer-close` *mix* [Bigloo Multimedia procedure]

Closes a mixer. The argument *mix* must be an instance of the `mixer` class.

`mixer-volume-get` *mix channel* [Bigloo Multimedia procedure]
`mixer-volume-set!` *mix channel leftv rightv* [Bigloo Multimedia procedure]

The function *mixer-volume-get* returns the left and right volume levels (two values) of the *channel* of the mixer *mix*. The *channel* is denoted by its name and is represented as a string of characters. The argument *mix* is an instance of the `mixer` class.

The function *mixer-volume-set!* changes the audio level of a mixer channel.

`soundcard::mixer` [Bigloo Multimedia class]

```
(class soundcard::mixer
   (device::bstring read-only))
```

The instances of the class `soundcard`, a subclass of the `mixer` class, are used to access physical soundcard as supported by operating systems. The class field *device* stands for the name of the system device (e.g., `"/dev/mixer"` for the Linux OS). During the initialization of the instance, the device is opened and initialized.

18.2.3 Playback

Bigloo supports various functions for playing music. These functions rely on two data structure: *music players* and *music status*. The first ones are used to control player backends. The second ones are used to get information about the music being played. The

following example shows how a simple music player using either MPlayer, MPG123, or MPC can be programmed with Bigloo.

```
(module musicplay
   (library multimedia)
   (main main))

(define (main args)
   (let ((files '())
 (backend 'mplayer)
 (command #f))
      (args-parse (cdr args)
 (("--mpg123" (help "Select the mpg123 back-end"))
  (set! backend 'mpg123))
 (("--mpc" (help "Select the mpc back-end"))
  (set! backend 'mpc))
 (("--mplayer" (help "Select the mplayer back-end"))
  (set! backend 'mplayer))
 (("--command" ?cmd (help "Set the command path"))
  (set! command cmd))
 (("--help" (help "This help"))
  (print "usage: music [options] file ...")
  (args-parse-usage #f)
  (exit 0))
 (else
  (set! files (cons else files))))
      ;; create a music player
      (let ((player (case backend
       ((mpg123)
(if command
    (instantiate::mpg123
       (path command))
    (instantiate::mpg123)))
       ((mplayer)
(if command
    (instantiate::mplayer
       (path command))
    (instantiate::mplayer)))
       ((mpc)
(instantiate::mpc)))))
         ;; fill the music play list
 (for-each (lambda (p) (music-playlist-add! player p)) (reverse files))
         ;; start playing
 (music-play player)
         ;; run an event loop with call-backs associated to some events
 (music-event-loop player
            :onstate (lambda (status)
                       (with-access::musicstatus status (state song volume)
                          (print "state   : " state)
                          (print "song    : " song)))
    :onmeta (lambda (meta)
       (print "meta    : " meta))
    :onvolume (lambda (volume)
       (print "volume  : " volume))))))
```

`music` [Bigloo Multimedia abstract class]

```
(abstract-class music
   (frequency::long (default 2000000))
```

This abstract class is the root class of all music players.

`musicproc::music` [Bigloo Multimedia class]

```
(class musicproc::music
   (charset::symbol (default 'ISO-LATIN-1)))
```

This class is used to reify player that are run in an external process.

`mplayer::musicproc` [Bigloo Multimedia class]

```
(class mplayer::musicproc
   (path::bstring read-only (default "mplayer"))
   (args::pair-nil read-only (default '("-vo" "null" "-quiet" "-slave" "-idle")))
   (ao::obj read-only (default #unspecified))
   (ac::obj read-only (default #unspecified)))
```

A player based on the external software `MPlayer`. Creating such a player spawns in background a MPlayer process.

`mpg123::musicproc` [Bigloo Multimedia class]

```
(class mpg123::musicproc
   (path::bstring read-only (default "mpg123"))
   (args::pair-nil read-only (default '("--remote"))))
```

A player based on the external software `mpg123`.

`mpc::music` [Bigloo Multimedia class]

```
(class mpc::music
   (hello read-only (default #f))
   (host read-only (default "localhost"))
   (port read-only (default 6600))
   (timeout read-only (default 10008993))
   (prefix (default #f)))
```

A MPC client.

- `hello`: an optional string written when the connection is establish with the MPD server.
- `prefix`: an optional path prefix to be removed from music playlist. This is needed because MPD can only play music files registered in is private database. The file names used by MPD are relative a root directory used to fill the database. The `prefix` field allows programmer to write portable code that manages play list file names independently of the player selected.

`musicstatus` [Bigloo Multimedia class]

```
(class musicstatus
   (state::symbol (default 'stop))
   (volume::obj (default -1))
   (repeat::bool (default #f))
   (random::bool (default #f))
   (playlistid::int (default -1))
   (playlistlength::int (default 0))
   (xfade::int (default 0))
   (song::int (default 0))
   (songid::int (default 0))
   (songpos (default 0))
   (songlength::int (default 0))
   (bitrate::int (default 0))
   (khz::int (default 0))
   (err::obj (default #f)))
```

The instances of the class `musicstatus` denote that state of a player.

`music-close` *music* [Bigloo Multimedia procedure]
`music-reset!` *music* [Bigloo Multimedia procedure]
`music-closed?` *music* [Bigloo Multimedia procedure]

Closes, resets, and tests the state of a music player.

`music-playlist-get` *music* [Bigloo Multimedia procedure]
`music-playlist-add!` *music song* [Bigloo Multimedia procedure]
`music-playlist-delete!` *music int* [Bigloo Multimedia procedure]
`music-playlist-clear!` *music* [Bigloo Multimedia procedure]

These functions controls the playlist used by a player.

Note: The *song* argument is an UTF8 encoded string (see Section Section 5.1.10 [Unicode (UCS-2) Strings], page 44) *whatever* the local file system encoding is. The function `music-playlist-get` returns a list of UTF8 encoded names.

- `music-playlist-get`: returns the list of songs (UTF8 names) of the current playlist.
- `music-playlist-add!`: adds an extra song (UTF8 name) at the end of the playlist.
- `music-delete!`: removes the song number *int* from the playlist.
- `music-clear!`: erases the whole playlist.

`music-play` *music* [*song*] [Bigloo Multimedia procedure]
`music-seek` *music time* [*song*] [Bigloo Multimedia procedure]
`music-stop` *music* [Bigloo Multimedia procedure]
`music-pause` *music* [Bigloo Multimedia procedure]
`music-next` *music* [Bigloo Multimedia procedure]
`music-prev` *music* [Bigloo Multimedia procedure]

These functions changes the state of the music player. The function `music-seek` seeks the playback position to the position *time*, which is an integer denoting a number of seconds.

`music-crossfade` *music int* [Bigloo Multimedia procedure]
`music-random-set!` *music bool* [Bigloo Multimedia procedure]
`music-repeat-set!` *music bool* [Bigloo Multimedia procedure]

These functions controls how songs playback should follow each other.

`music-volume-get` *music* [Bigloo Multimedia procedure]
`music-volume-set!` *music vol* [Bigloo Multimedia procedure]

Get and set the audio volume of a player. Some player use the native mixer supported by the operating system some others use a software mixer unrelated to the hardware.

`music-status` *music* [Bigloo Multimedia procedure]
`music-update-status!` *music status* [Bigloo Multimedia procedure]

The function `music-status` returns an instance of the `musicstatus` class which denotes the state of the player. The function `music-update-status!` updates this status.

`music-song` *music* [Bigloo Multimedia procedure]
`music-songpos` *music* [Bigloo Multimedia procedure]

These two functions return the number of the song being played and the position in the song. These functions are somehow redundant with the function `music-status` because the status also contains information about the playback song and playback position. However, for some players getting the music song and the playback position is cheaper than getting the whole player status.

`music-meta` *music* [Bigloo Multimedia procedure]

Returns the metadata the current song.

`music-reset-error!` *music* [Bigloo Multimedia procedure]

Reset the previous errors detected by a player.

`music-event-loop` *music* [*:onstate*] [*:onmeta*] [*:onerror*] [*:onvolume*] [Bigloo Multimedia procedure]

The function `music-event-loop` enable event notifications when the state of a player changes. The keyword arguments are:

- `:onstate`, a function of one parameter. When the player state changes, this function is called with an instance of `musicstatus` as first actual parameter.
- `:onmeta`, a function of two parameters. This function is called when a metadata is detected in the music currently played.
- `:onerror`, a function of one parameter, invoked when an error is detected.
- `:onvolume`, a function of one parameter, invoked when the volume changes.

18.2.4 Music Player Daemon

Music Player Daemon (MPD in short) allows remote access for playing music `http://www.musicpd.org`. MPD is designed for integrating a computer into a stereo system that provides control for music playback over a local network. The Bigloo class `mpc` implements a `mpd` client. All Bigloo players can be access via the MPD protocol, using the

The following example shows how to access a MPlayer music player using the MPD protocol with a simple Bigloo program:

```
(module mpd
   (library multimedia pthread)
   (main main))

(define (main argv)
   (let ((db (instantiate::mpd-database
(directories (cdr argv))))
 (serv (make-server-socket 6600))
 (music (instantiate::mplayer)))
      (let loop ()
 (thread-start! (make-mpd-connection-thread music db sock))
 (loop))))

(define (make-mpd-connection-thread music db sock)
   (instantiate::pthread
      (body (lambda ()
       (let ((pi (socket-input sock))
     (po (socket-output sock)))
```

```
(input-timeout-set! pi 10000)
(output-timeout-set! po 10000)
(unwind-protect
   (mpd music pi po db)
   (socket-close sock)))))))
```

`mpd` *music input-port output-port database* [*:log*] [Bigloo Multimedia procedure]

The function `mpd` implements a MPD server. It reads commands from the *input-port* and write results to *output-port*. The argument *database*, an instance of the `mpd-database` class, describes the music material that can be delivered by this player.

`mpd-database` [Bigloo Multimedia class]

```
(class mpd-database
   (directories::pair-nil read-only)
```

The field `directories` contains the list of the directories that contains music files.

18.3 Color

The multimedia library provides functions for dealing with colors.

`hsv->rgb` *h s v* [Bigloo Multimedia procedure]
`hsl->rgb` *h s l* [Bigloo Multimedia procedure]
`rgb-hsv` *r g b* [Bigloo Multimedia procedure]
`rgb-hsl` *r g b* [Bigloo Multimedia procedure]

These procedures converts from and to HSV, HSL, and RGB representations. The argument *h* is an integer in the range `[0..360]`, the arguments *s*, *v*, and *l* in the range `[0..100]`. The arguments *r*, *g*, and *b* are in the range `[0..255]`. These procedures returns multiple-values.

```
(multiple-value-bind (r g b)
   (hsv->rgb 340 34 56)
   (list r g b)) ⇒ (143 94 110)
(multiple-value-bind (h s v)
   (rgb->hsv 255 0 0)
   (list h s v)) ⇒ (0 100 100)
```

19 Mail

Bigloo provides various facilities for handling mails. It provides facilities for parsing many formats commonly used in composing mails (quoted printable, vcard, mime types). It also provides facilities for dealing with mail servers. For that it proposes an abstracted view of mail servers with two implementations: `imap` and `maildir`.

19.1 RFC 2045 – MIME, Part one

This section described the functions offered by Bigloo to encode and decode some of the formats specified in the RFC 2045 `http://tools.ietf.org/html/rfc2045`.

`quoted-printable-encode` *string* [Bigloo Mail procedure]
`quoted-printable-decode` *string* [Bigloo Mail procedure]

These functions encode/decode a string into and from the `quoted-printable` format.

Examples:

```
(quoted-printable-encode "foo bar") ⇒ "foobar=20"
(quoted-printable-decode "foobar=20") ⇒ "foo bar"
```

`quoted-printable-encode-port` *ip op* [Bigloo Mail procedure]
`quoted-printable-decode-port` *ip op* [*rfc2047*] [Bigloo Mail procedure]

These functions are similar to `quoted-printable-encode` and `quoted-printable-decode` except that they operate on `input-port`s and `output-port`s.

The function `quoted-printable-decode-port` accepts an optional argument: *rfc2047*. If this argument is `#t`, then the parsing stops on the prefix `?=`, which is a marker in the mail subject as specified by the RFC 2047, (see `http://tools.ietf.org/html/rfc2047`) is found.

`mime-content-decode` *string* [Bigloo Mail procedure]
`mime-content-decode-port` *input-port* [Bigloo Mail procedure]

These two functions parse respectively a `string` and an `input-port` and return a list of three elements:

- a content type,
- a content subtype,
- options.

Example:

```
(mime-content-type-decode "text/plain; boundary=Apple-Mail-11")
  ⇒ (text plain ((boundary . Apple-Mail-11)))
```

`mime-content-disposition-decode` *string* [Bigloo Mail procedure]
`mime-content-disposition-decode-port` *input-port* [Bigloo Mail procedure]

These two functions parse respectively a `string` and an `input-port` and return a list describing the content disposition.

Example:

```
(mime-content-disposition-decode "attachment; filename=\"smine.p7s\"")
  ⇒ (attachment ((filename . smine.p7s)))
```

`mime-multipart-decode` *string boundary* [*recursive*] [Bigloo Mail procedure]
`mime-multipart-decode-port` *input-port boundary* [*recursive*] [Bigloo Mail procedure]

These two functions parse respectively a `string` and an `input-port` and return a list of mime sections.

If the optional argument *recursive* controls whether subparts of a multipart section must be decoded are not. If the *recursive* is `#t` then all subparts of the multipart content are decoded. The result is a fully decoded multipart section. If *recursive* is `#f` subparts are not decoded and included in the result as plain strings.

19.2 RFC 2047 – MIME, Part three

This section described the function offered by Bigloo to decode the RFC 2047 encoding used in mail headers (see `http://tools.ietf.org/html/rfc2047`).

`rfc2047-decode-port` *ip op* [*:charset iso-latin-1*] [Bigloo Mail procedure]
`rfc2047-decode` *string* [*:charset iso-latin-1*] [Bigloo Mail procedure]

These functions decode mail header fields encoded using the RFC 2047 specification. The optional argument *charset* specified in which charset the result should be encoded. The allowed values are:

- `utf-8`
- `iso-latin-1`
- `cp-1252`

Example:

```
(map char->integer
  (string->list (rfc2047-decode "Poste =?ISO-8859-1?Q?t=E9l=E9phonique?=")))
  ⇒ (80 111 115 116 101 32 116 233 108 233 112 104 111 110 105 113 117 101)
(string-for-read (rfc2047-decode "Poste =?ISO-8859-1?Q?t=E9l=E9phonique?=" :charset 'utf8))
  ⇒ "Poste t\303\251l\303\251phonique"
```

19.3 RFC 2426 – MIME, Part three

This section presents the facilities supported by Bigloo for dealing with `vcard`s.

`vcard` [Bigloo Mail class]

```
(class vcard
  (version::bstring (default "2.1"))
  (fn (default #f))
  (familyname (default #f))
  (firstname (default #f))
  (face (default #f))
  (url (default #f))
  (org (default #f))
  (emails::pair-nil (default '()))
  (phones::pair-nil (default '()))
  (addresses::pair-nil (default '())))
```

The class `vard` is used to reify in memory a vcard as parsed by the function `port->vcard` and `string->vcard`.

Except `emails`, `phones`, and `addresses`, all fields are optional. They should be either `#f` or a string.

- `face` is a flat list of strings.
- `phones` is an alist whose elements are pairs of two strings.
- `addresses` is a list composed of:
  - the postoffice, a string,
  - a list of strings denoting the street address,
  - a string denoting the city,
  - a string denoting the region,
  - a string denoting the zip code,
  - a string denoting the zip country.

  All street values are required and must be provided. The empty string should be used to denote empty values.

`port->vcard::vcard` *ip* [*:charset-encoder*] [Bigloo Mail function]
`string->vcard::vcard` *str* [*:charset-encoder*] [Bigloo Mail function]

These two functions parse a *vcard* to produce a `vcard` instance. The optional argument *charset-encoder*, when provided, must be a function of argument: a string to be decoded. Vcard strings are UTF-8 encoded. The *charset-encoder* can be used to encode on-the-fly the strings found in the vcard in a difference encoding.

19.4 RFC 2822 – Internet Message Format

This section described the functions offered by Bigloo to encode and decode some of the formats specified in the RFC 2822 (`http://tools.ietf.org/html/rfc2045`). It mainly supports functions for parsing email headers and for decoding email addresses.

`mail-header->list` *obj* [Bigloo Mail procedure]

The function `mail-header->list` parses a mail header that can either be implemented as a string or an input port. It returns a list of fields.

Example:

```
(mail-header->list "Return-Path: <foo.bar@inria.fr>
Received: from eurus.inria.fr ([unix socket])")
  ⇒
  ((return-path . "<foo.bar@inria.fr>") (received . "from eurus.inria.fr ([unix socket])"))
```

`email-normalize` *string* [Bigloo Mail procedure]

The function `email-normalize` extracts the actual email address from an email representation.

Example:

```
(email-normalize "foo bar <foo.bar@inria.fr>") ⇒ "foo.bar@inria.fr"
```

`rfc2822-address-display-name` *string* [Bigloo Mail procedure]

Extract the name component of an email.

Example:

```
(rfc2822-address-display-name "Foo Bar <foo.bar@inria.fr>") ⇒ "Foo Bar"
(rfc2822-address-display-name "<foo.bar@inria.fr>") ⇒ "foo bar"
```

19.5 Mail servers – imap and maildir

Bigloo implements the `imap` protocol (`http://tools.ietf.org/html/rfc3501`) and the `maildir` format. This section presents the API for manipulating them both.

19.5.1 Mailboxes

`mailbox` [Bigloo Mail class]

```
(abstract-class mailbox
  (label::bstring (default "")))
```

The abstract class `mailbox` is the common ancestors to all the mailbox implementations. It allows the definitions of various generic functions that deal with mail messages and mail folders.

`&mailbox-error` [Bigloo Mail class]

```
(abstract-class &mailbox-error::&error)
```

The `&mailbox-error` is the super class of all the errors that can be raised when accessing mail servers, except the parsing errors that inherit from the `&parse-error` super class.

`mailbox-close` *mailbox* [Bigloo Mail procedure]

Close the mailbox connection.

Example:

```
(let ((mbox (if (network-up?)
                (instantiate::imap (socket ...))
                (instantiate::maildir (path my-local-cache)))))
  (mailbox-close mbox))
```

`mailbox-separator` *mailbox* [Bigloo Mail procedure]

Returns a string denoting the separator (commonly `"` or `.`) used by the *mailbox*.

`mailbox-prefix` *mailbox* [Bigloo Mail procedure]

Returns the prefix of the *mailbox*, a string or `#f`.

`mailbox-hostname` *mailbox* [Bigloo Mail procedure]

Returns the hostname of the *mailbox*, a string or `#f`.

`mailbox-folders` *mailbox* [Bigloo Mail procedure]

Returns a list of strings denoting the folder names of the *mailbox*.

`mailbox-folder-select!` *mailbox string* [Bigloo Mail procedure]

Selects one folder of the *mailbox*. This function is central to mailboxes because all messages are referenced relatively to the folder selection. All the functions that operates on `uid` implicitly access the current folder selection.

`mailbox-folder-unselect!` *mailbox* [Bigloo Mail procedure]

Unselects the *mailbox* current selected folder.

`mailbox-folder-create!` *mailbox folder* [Bigloo Mail procedure]

Creates a new *folder* denotes by a fully qualified name.

Example

```
(mailbox-create! mbox "INBOX.scheme.bigloo")
```

`mailbox-folder-delete!` *mailbox folder* [Bigloo Mail procedure]
Deletes an empty *folder*.

`mailbox-folder-rename!` *mailbox old new* [Bigloo Mail procedure]
Renames a folder.

`mailbox-folder-move!` *mailbox folder dest* [Bigloo Mail procedure]
Moves the *folder* into the destination folder *dest*.

`mailbox-subscribe!` *mailbox folder* [Bigloo Mail procedure]
`mailbox-unsubscribe!` *mailbox folder* [Bigloo Mail procedure]
Subscribe/unsubscribe to a folder. This allows `imap` servers not to present the entire list of folders. Only subscribed folders are returned by `mailbox-folders`. These functions have no effect on `maildir` servers.

`mailbox-folder-exists?` *mailbox folder* [Bigloo Mail procedure]
Returns `#t` if and only if *folder* exists in *mailbox*. Returns `#f` otherwise.

`mailbox-folder-status` *mailbox folder* [Bigloo Mail procedure]
Returns the status of the *folder*. A status is an alist made of the number of unseen mail, the uid validity information, the uid next value, the number of recent messages, and the overall number of messages.

`mailbox-folder-uids` *mailbox* [Bigloo Mail procedure]
Returns the list of UIDs (a list of integers) of the messages contained in the currently selected folder.

`mailbox-folder-dates` *mailbox* [Bigloo Mail procedure]
Returns the list of dates of the messages contained in the currently selected folder.

`mailbox-folder-delete-messages!` *mailbox* [Bigloo Mail procedure]
Deletes the messages marked as *deleted* of the currently selected folder.

`mailbox-folder-header-fields` *mailbox field* [Bigloo Mail procedure]
Returns the list of headers *fields* of the message of the current folder.

`mailbox-message` *mailbox uid* [Bigloo Mail procedure]
Returns the message *uid* in the current folder.

`mailbox-message-path` *mailbox uid* [Bigloo Mail procedure]
Returns the full path name of the message *uid*.

`mailbox-message-body` *mailbox uid* [*len*] [Bigloo Mail procedure]
Returns the body of the message *uid*. If *len* is provided, only returns the first *len* characters of the body.

`mailbox-message-header` *mailbox uid* [Bigloo Mail procedure]
Returns the header as a string of the message *uid*.

`mailbox-message-header-list` *mailbox uid* [Bigloo Mail procedure]
Returns the header as an alist of the message *uid*.

`mailbox-message-header-field` *mailbox uid field* [Bigloo Mail procedure]
Extracts one field from the message header.

`mailbox-message-size` *mailbox uid* [Bigloo Mail procedure]
Returns the size of the message.

`mailbox-message-info` *mailbox uid* [Bigloo Mail procedure]
Returns the information relative to the message *uid*. This a list containing the message identifier, its uid, the message date, the message size, and the message flags.

`mailbox-message-flags` *mailbox uid* [Bigloo Mail procedure]
`mailbox-message-flags-set!` *mailbox uid lst* [Bigloo Mail procedure]
Sets/Gets the flags of the message *uid*. This is a list of strings. Typical flags are:

- `\Flagged`
- `\Answered`
- `\Deleted`
- `\Seen`

`mailbox-message-delete!` *mailbox uid* [Bigloo Mail procedure]
Deletes the message *uid*.

`mailbox-message-move!` *mailbox uid folder* [Bigloo Mail procedure]
Moves the message *uid* into the new *folder* (denoted by a string).

`mailbox-message-create!` *mailbox folder content* [Bigloo Mail procedure]
Creates a new message in the *folder* whose content is given the string *content*.

19.5.2 IMAP (RFC 3501)

`imap` [Bigloo Mail class]

```
(class imap::mailbox
  (socket::socket read-only))
(define mbox
  (instantiate::maildir
    (label "My Remote Mailbox")
    (socket (imap-login (make-client-socket "imap.inria.fr" 993)
                        "serrano" "XXX"))))
```

`&imap-parse-error` [Bigloo Mail class]

```
(class &imap-parse-error::&io-parse-error)
```

`&imap-error` [Bigloo Mail class]

```
(class &imap-error::&mailbox-error)
```

`imap-login` *socket user password* [Bigloo Mail procedure]
Log a user into an imap server. The *socket* must have been created first. The argument *user* is a string and denotes the user name. The argument *password* is a string too and it contains the user password. This function returns as value the *socket* it has received. If the operation fails the function raises a `&imap-error` exception.

Example:

(define mbox (imap-login (make-client-socket "imap.inria.fr" 993 :timeout 200000) "serrano" "XXX"))

(print (mailbox-folders mbox))

`imap-logout` *socket* [Bigloo Mail procedure]
Closes an `imap` connection.

`imap-capability` *socket* [Bigloo Mail procedure]
Returns the list of capabilities supported the `imap` server.

19.5.3 Maildir

`maildir` [Bigloo Mail class]

```
(class maildir::mailbox
  (prefix::bstring read-only (default "INBOX"))
  (path::bstring read-only))
```

Example:

```
(define mbox
  (instantiate::maildir
    (label "My Mailbox")
    (path (make-file-name (getenv "HOME") ".maildir"))))

(tprint (mailbox-folders mbox))
```

`&maildir-error` [Bigloo Mail class]

```
(class &maildir-error::&mailbox-error)
```

20 Text

This chapter describes the Bigloo API for processing texts.

20.1 BibTeX

`bibtex` *obj* [Bigloo Text function]
`bibtex-port` *input-port* [Bigloo Text function]
`bibtex-file` *file-name* [Bigloo Text function]
`bibtex-string` *string* [Bigloo Text function]

These function parse BibTeX sources. The variable *obj* can either be an input-port or a string which denotes a file name. It returns a list of BibTeX entries.

The functions `bibtex-port`, `bibtex-file`, and `bibtex-string` are mere wrappers that invoke `bibtex`.

Example:

```
(bibtex (open-input-string "@book{ as:sicp,
  author  = {Abelson, H. and Sussman, G.},
  title  = {Structure and Interpretation of Computer Programs},
  year  = 1985,
  publisher  = {MIT Press},
  address  = {Cambridge, Mass., USA},
}")) ⇒ (("as:sicp" BOOK
                  (author ("Abelson" "H.") ("Sussman" "G."))
                  (title . "Structure and Interpretation of Computer Programs")
                  (year . "1985")
                  (publisher . "MIT Press")
                  (address . "Cambridge, Mass., USA")))
```

`bibtex-parse-authors` *string* [Bigloo Text function]

This function parses the author field of a bibtex entry.

Example:

```
(bibtex-parse-authors "Abelson, H. and Sussman, G.")
⇒ (("Abelson" "H.") ("Sussman" "G."))
```

20.2 Character strings

`hyphenate` *word hyphens* [Bigloo Text function]

The function `hyphenate` accepts as input a single word and returns as output a list of subwords. The argument *hyphens* is an opaque data structure obtained by calling the function `load-hyphens` or `make-hyphens`.

Example:

```
(hyphenate "software" (load-hyphens 'en)) ⇒ ("soft" "ware")
```

`load-hyphens` *obj* [Bigloo Text function]

Loads an hyphens table and returns a data structure suitable for `hyphenate`. The variable *obj* can either be a file name containing an hyphens table or a symbol denoting a pre-defined hyphens table. Currently, Bigloo supports two tables: `en` for an English table and `fr` for a French table. The procedure `load-hyphens` invokes `make-hyphens` to build the hyphens table.

Example:

```
(define (hyphenate-text text lang)
   (let ((table (with-handler
                   (lambda (e)
                      (unless (&io-file-not-found-error? e)
                         (raise e)))
                   (load-hyphens lang)))
         (words (string-split text " ")))
      (if table
          (append-map (lambda (w) (hyphenate w table)) words)
          words)))
```

The procedure `hyphenate-text` hyphenates the words of the `text` according to the rules for the language denoted by its code `lang` if there is a file *lang*`-hyphens.sch`. If there is no such file, the text remains un-hyphenated.

`make-hyphens` [*:language*] [*:exceptions*] [*:patterns*] [Bigloo Text function]

Creates an hyphens table out of the arguments *exceptions* and *patterns*.

The implementation of the table of hyphens created by `make-hyphens` follows closely Frank Liang's algorithm as published in his doctoral dissertation *Word Hy-phen-a-tion By Com-pu-ter* available on the TeX Users Group site here: `http://www.tug.org/docs/liang/`. This table is a *trie* (see `http://en.wikipedia.org/wiki/Trie` for a definition and an explanation).

Most of this implementation is borrowed from Phil Bewig's work available here: `http://sites.google.com/site/schemephil/`, along with his paper describing the program from which the Bigloo implementation is largely borrowed.

exceptions must be a non-empty list of explicitly hyphenated words.

Explicitly hyphenated words are like the following: `"as-so-ciate"`, `"as-so-ciates"`, `"dec-li-na-tion"`, where the hyphens indicate the places where hyphenation is allowed. The words in *exceptions* are used to generate hyphenation patterns, which are added to *patterns* (see next paragraph).

patterns must be a non-empty list of hyphenation patterns.

Hyphenation patterns are strings of the form `".anti5s"`, where a period denotes the beginning or the end of a word, an odd number denotes a place where hyphenation is allowed, and an even number a place where hyphenation is forbidden. This notation is part of Frank Liang's algorithm created for Donald Knuth's TeX typographic system.

20.3 Character encodings

`gb2312->ucs2` *string* [Bigloo Text function]

Converts a GB2312 (aka cp936) encoded 8bits string into an UCS2 string.

21 CSV

This chapter describes the Bigloo API for processing CSV spreadsheets. This chapter has been written by Joseph Donaldson, as the implementation of the CSV library.

21.1 Overview

The Bigloo csv library supports the parsing of csv and csv-like data. By default, it enables the parsing of comma, tab, and pipe separated data. In addition, facilities are provided that enable extending the library to support additonal csv-like formats.

The rest of this document describes the Bigloo csv application programming interface.

21.2 API Reference

`read-csv-record` *input-port* [*custom-lexer*] [bigloo procedure]
`read-csv-record` has one required argument, the input-port of the csv data to parse, and an optional argument indicating the lexer to use, by default the lexer supporting standard csv files. It returns a single record, as a list, or `#eof-object`. Upon error, it will throw an `&invalid-port-error` or `&io-parse-error` exception.

`read-csv-records` *input-port* [*custom-lexer*] [bigloo procedure]
`read-csv-records` has one required argument, the input-port of the csv data to parse, and an optional argument indicating the lexer to use, by default the lexer supporting standard csv files. It returns all of the records, as a list of lists, or `#eof-object`. Upon error, it will throw an `&invalid-port-error` or `&io-parse-error exception`.

`csv-for-each` *proc input-port* [*custom-lexer*] [bigloo procedure]
`csv-for-each` has two required arguments, a procedure to apply to each record and the input-port of the csv data to parse, and an optional argument indicating the lexer to use, by default the lexer supporting standard csv files. It returns `#unspecified`. Upon error, it will throw an `&invalid-port-error` or `&io-parse-error exception`.

`csv-map` *proc input-port* [*custom-lexer*] [bigloo procedure]
`csv-map` has two required arguments, a procedure to apply to each record and the input-port of the csv data to parse, and an optional argument indicating the lexer to use, by default the lexer supporting standard csv files. It returnsthe results of applying `proc` to each record as a list. Upon error, it will throw an `&invalid-port-error` or `&io-parse-error` exception.

`make-csv-lexer` *sep quot* [bigloo form]
`make-csv-lexer` has two required arguments, a character used to separate records and a character for quoting. It returns custom lexer.

`bigloo variable` +*csv-lexer*+ [Variable]
+*csv-lexer*+ is a bigloo-csv lexer supporting the standard comma-separated value format.

`bigloo variable` +*tsv-lexer*+ [Variable]
+*tsv-lexer*+ is a bigloo-csv lexer supporting the tab-separated value format.

`bigloo variable` +*psv-lexer*+ [Variable]

+*psv-lexer*+ is a bigloo-csv lexer supporting the pipe-separated value format.

The following is a simple example of using the bigloo-csv library. It parses a single record from the given csv data and prints it.

```
(module example
   (library bigloo-csv)
   (main main))

(define +csv-data+ "dog,cat,horse\npig,cow,squirrel")

(define (main args)
   (let ((in (open-input-string +csv-data+)))
      (unwind-protect
         (print (read-csv-record in))
         (close-input-port in))))
```

22 Eval and code interpretation

This chapter describes the Bigloo evaluator.

22.1 Eval compliance

Bigloo includes an interpreter. Unfortunately, the language accepted by the interpreter is a proper subset of that accepted by the compiler. The main differences are:

- No foreign objects can be handled by interpreter.
- Classes of the object system cannot be declared within interpreted code.
- The interpreter ignores modules, and has a unique global environment.

Compiled code and interpreted code can be mixed together. That is, interpreted code is allowed to call compiled code and vice versa. This connection can be use to circumvent the missing features of the interpreter (see Section see Section 2.2 [Module Declaration], page 7, for a description of how to connect compiled and interpreted code).

By default the evaluator assumes that operators from the standard library (e.g., `+`, `car`) are immutable. Hence, it optimizes these operators's calls. This optimization can be disabled using the `bigloo-eval-strict-module` parameter described in the chapter describing the parameters (see see Chapter 24 [Parameters], page 231).

22.2 Eval standard functions

`eval` *exp* [*env*] [procedure]

This form evaluates *exp*. The second argument is optional. It can be the evaluation of one of these three function forms:

```
(scheme-report-environment 5)
(null-environment 5)
(interaction-environment)
```

`scheme-report-environment` *version* [procedure]
`null-environment` *version* [procedure]
`interaction-environment` *version* [procedure]

These three procedures have the definitions given in the R5RS so see Section "6.5 Eval" in *R5RS*, for more details.

`byte-code-compile` *exp* [*env* (*default-environment*)] [bigloo procedure]
`byte-code-run` *byte-code* [bigloo procedure]

The function `byte-code-compile` compiles a Scheme expression into a sequence of byte codes that is implemented as a string. The function `byte-code-run` execute such a sequence.

`repl` [bigloo procedure]

This invokes the *read-eval-print* loop. Several `repl` can be embedded.

The `repl` function can be used to implement custom Bigloo interpreters. For instance, one may write:

```
(module repl)
(repl)
```

When compiled, this will deliver an executable containing the sole Bigloo interpreter.

`set-repl-error-notifier!` *notifier* [bigloo procedure]
`get-repl-error-notifier` [bigloo procedure]

Set or get the procedure used in the REPLs to display errors. The notifier is a procedure of one argument, the error or exception that has been raised. Example:

```
(set-repl-error-notifier!
   (lambda (e)
      (print "=== error ======)
      (exception-notify e)))
```

`quit` [bigloo procedure]

This exits from the currently running `repl`. If the current `repl` is the first one then this function ends the interpreter.

`set-prompter!` *proc* [bigloo procedure]

The argument *proc* has to be a procedure of one argument and invoking this function sets the `repl` prompter. That is, to display its prompt, `repl` invokes *proc* giving it the nesting level of the current loop as its argument.

`get-prompter` [bigloo procedure]

Returns the current `repl` prompter.

`set-repl-printer!` *proc* [bigloo procedure]

The argument *proc* has to be a procedure accepting one or two arguments. This function sets the `repl` display function. That is, to display the result of its evaluations, `repl` invokes *proc* giving it the evaluated expression as first argument and the current output port (or a file in case of transcript) as second argument. `Set-repl-printer!` returns the former `repl` display function.

For instance, one may write:

```
1:=> (define x (cons 1 2))          ⊣ X
1:=> (define y (cons x x))          ⊣ Y
1:=> y                              ⊣ (#0=(1 . 2) . #0#)
1:=> (set-repl-printer! display)    ⊣ #<procedure:83b8c70.-2>
1:=> y                              ⊣ ((1 . 2) 1 . 2)
```

`native-repl-printer` [bigloo procedure]

Returns the native (default) `repl` display function.

`expand` *exp* [bigloo procedure]

Returns the value of *exp* after all macro expansions have been performed.

`expand-once` *exp* [bigloo procedure]

Returns the value of *exp* after one macro expansion has been performed.

It is possible to specify files which have to be loaded when the interpreter is invoked. For this, see section see Chapter 31 [Compiler Description], page 275.

If a Bigloo file starts with the line:

```
#! bigloo-command-name
```

and if this file is executable (in the meaning of the system) and if the user tries to execute it, Bigloo will evaluate it. Note also that SRFI-22 support enables to run any Unix interpreter (see Chapter 30 [SRFIs], page 269).

`load` *filename* [bigloo procedure]
`loadq` *filename* [bigloo procedure]

Filename should be a string naming an existing file which contains Bigloo source code. This file is searched in the current directory and in all the directories mentioned in the variable `*load-path*`. The `load` procedure reads expressions and definitions from the file, evaluating them sequentially. If the file loaded is a module (i.e. if it begins with a regular module clause), load behaves as module initialization. Otherwise, this function returns the result of the last evaluation. The function `loadq` differs from the function `load` in the sense that `loadq` does not print any intermediate evaluations.

Both functions return the full path of the loaded file.

`loada` *filename* [bigloo procedure]

Loads an "access file", which allows the interpreter to find the modules imported by a loaded module. It returns the full path of the loaded file.

`*load-path*` [bigloo variable]

A list of search paths for the `load` functions.

`dynamic-load` *filename #!optional* (*init init-point*) [bigloo procedure]

Loads a shared library named `filename`. Returns the value of the last top-level expression.

Important note: The function `dynamic-load` can only be used from compiled modules linked against dynamic libraries. In particular, the `dynamic-load` function can be issued from the `bigloo` command if and only if the option `--sharedcompiler=yes` has been used when configuring Bigloo. If the `bigloo` command is not linked against dynamic libraries and if `dynamic-load` is required inside a read-eval-print loop (REPL) it exists a simple workaround. It consists in implementing a new REPL and linking it against dynamic libraries. This can be done as:

```
$ cat > new-repl.scm <<EOF
(module new-repl)
(repl)
EOF
$ bigloo new-repl.scm -o new-repl
$ new-repl
1:=> (dynamic-load ...)
```

If *init-point* is specified and if it is a string and if the library defines a function named `init-point`, this function is called when the library is loaded. *Init-point* is a C identifier, not a Scheme identifier. In order to set the C name a Scheme function, use the extern `export` clause (see Section see Chapter 26 [C Interface], page 235). If the *init-point* is provided and is not a string, no initialization function is called after the library is loaded. If the *init-point* value is not provided, once the library is loaded, `dynamic-load` uses the Bigloo default entry point. Normally you should *not* provide an *init-point* to `dynamic-load` unless you known what you are doing. When producing C code, to force the Bigloo compiler to emit such a default entry point, use the `-dload-sym` compilation option (see Section see Chapter 31 [Compiler

Description], page 275). This option is useless when using the JVM code generator. Let's assume a Linux system and two Bigloo modules. The first:

```
(module mod1
   (eval (export foo))
   (export (foo x)))

(define (foo x)
   (print "foo: " x))

(foo 4)
```

The second:

```
(module mod2
   (import (mod1 "mod1.scm"))
   (eval (export bar))
   (export (bar x)))

(define (bar x)
   (print "bar: " x))

(bar 5)
```

If these modules are compiled as:

```
$ bigloo mod1.scm -c -o mod1.o
$ bigloo mod2.scm -c -o mod2.o -dload-sym
```

Then, if a shared library is built using these two modules (note that on non Linux systems, a different command line is required):

```
$ ld -G -o lib.so mod1.o mod2.o
```

Then, `lib.so` cant be dynamically loaded and the variables it defines used such as :

```
$ bigloo -i
(dynamic-load "lib.so")
     ⊣ foo: 4
       bar: 5
1:=> (foo 6)
     ⊣ foo: 7
```

As the example illustrates, when Bigloo modules are dynamically loaded, they are initialized. This initialization is ensure *only* if `dynamic-load` is called with exactly one parameter. If `dynamic-load` is called with two parameters, it is of the responsibility of the program to initialize the dynamically loaded module before using any Scheme reference.

Note: In order to let the loaded module accesses the variables defined by the loader application, special compilation flags must be used (e.g., `-rdynamic` under the Linux operating system). `Dynamic-load` is implemented on the top of the `dlopen` facility. For more information read the `dlopen` and `ld` manuals.

`dynamic-unload` *filename* [bigloo procedure]

On the operating system that supports this facility, unloads a shared library. Returns `#t` on success. Returns `#f` otherwise.

`*dynamic-load-path*` [bigloo variable]

A list of search paths for the `dynamic-load` functions.

`dynamic-load-symbol` *filename name #!optional module* [bigloo-procedure]
`dynamic-load-symbol-get` *dlsym* [bigloo-procedure]
`dynamic-load-symbol-set` *dlsym val* [bigloo-procedure]

The function `dynamic-load-symbol` looks up for a variable in the dynamic library *filename*. If found, it returns a `custom` Bigloo object denoting that variable. Otherwise it returns `#f`. This function assumes that *filename* has previously been successfully loaded with `dynamic-load`. If not, an error is raised. The argument *filename* must be equal (in the sense of `string=?` to the argument used to load the library.

The C name of the looked up variable is *name* is *module* is not provided. Otherwise, it is the result of calling `bigloo-module-mangle` with *name* and *module* as arguments.

The function `dynamic-load-symbol-get` returns the value of a dynamically loaded variable.

The function `dynamic-load-symbol-set` sets the value of a dynamic loaded variable. It assumes that the variable is writable, i.e., that it has not been compiled as a C constant.

`transcript-on` *filename* [procedure]
`transcript-off` [procedure]

22.3 Eval command line options

This section presents the Bigloo compiler options that impact the interaction between compiled and interpreted code. The whole list of the Bigloo compiler options can be found in Chapter 31 [The Bigloo command line], page 275.

- `-i` Don't compile a module, interpret it!
- `-export-all` Make all the bindings *defined* by the compiled module available from the interpreter.
- `-export-export` Make all the bindings *exported* by the compiled module available from the interpreter.
- `-export-mutable` Make all the bindings *exported* by the compiled module mutable from outside the module. This option is *dangerous*! Either all the modules composing the application must be compiled with or without `-export-mutable`. It is impossible to mix `-export-mutable` enabled and disabled compilations.

22.4 Eval and the foreign interface

To be able to get access to foreign functions within the Bigloo interpreter, some extra measurements have to be taken. The foreign functions have to be present in the interpreter binary, which means you have to compile a custom interpreter. This is described in Section Section 26.5 [Using C bindings within the interpreter], page 246.

23 Macro expansion

Bigloo makes use of two macro expansion system. The one based on the expansion passing style [Dybvig et al. 86] and the one advocated by the R5RS, for which see [No value for "[R5RS]"].

23.1 Expansion passing style macros

`define-expander` *name proc* [bigloo syntax]

This form defines an expander, *name*, where *proc* is a procedure of two arguments: a form to macro-expand, and an expander.

`define-macro` (*name* [*args*]...) *body* [bigloo syntax]

This form is itself macro-expanded into a `define-expander` form.

Macro expanders cannot be exported or imported since there is no way to specify expanders in a module declaration.

Macros defined with `define-expander` and `define-macro` are used by both the compiler and the interpreter.

Here is an example of an expander:

```
(define-expander when
   (lambda (x e)
      (match-case x
         ((?- ?test . ?exps)
          (e `(if ,test (begin ,@exps)) e))
         (else
           (error "when" "illegal form" x)))))

(when (> a 0) (print a) a)
   ↦ (if (> a 0) (begin (print a) a))
```

The same example can written with a `define-macro` form:

```
(define-macro (when test . exps)
   `(if ,test (begin ,@exps)))
```

23.2 Revised(5) macro expansion

Bigloo support the Revised(5) Report on the Scheme programming language. For a detailed documentation see See Section "r5rs.info" in `R5RS`.

`let-syntax` (*binding*...) *body* [syntax]
`letrec-syntax` (*binding*...) *body* [syntax]
`define-syntax` *keyword transformer* [syntax]
`syntax-rules` *literals rule*... [syntax]

These three forms are compatible with the description of the Revised(5) Report on the Algorithmic Language Scheme.

Implementation Note: Current Bigloo does not ensure hygiene for `let-syntax` and `letrec-syntax`. Hygienic expansion is only guaranteed for `define-syntax`.

24 Parameters

The Bigloo parameters drive the global behavior programs. A parameter is accessed via a pair of functions: a reader and a setter. The type of the value is given, in this documentation, by the name of the parameter of the setter.

`bigloo-strict-r5rs-strings` [bigloo function]
`bigloo-strict-r5rs-strings-set!` *boolean* [bigloo function]

Traditional syntax conforms to the Revised Report if the parameter `bigloo-strict-r5rs-strings` is not `#f`. Otherwise constant strings specified by the `"([^"]|\")*"` are considered as foreign strings.

For example, after reading the expression `"1\n23\t4\"5"`, the following string is built, which is equal to `(string #\1 #\n #\2 #\3 #\t #\4 #\" #\5)` if `(bigloo-strict-r5rs-strings)` is not `#f`. It is `(string #\1 #\n #\2 #\3 #\tab #\4 #\" #\5)` otherwise.

Printing this string will produce: `1n23t4"5`.

The new foreign syntax allows C escape sequences to be recognized. For example, the expression `#"1\n23\t4\"5"` builds a string equal to:

```
(string #\1 #\newline #\2 #\3 #\t #\4 #\" #\5)
```

and printing this string will then produce:

```
1
23    4"5
```

`bigloo-compiler-debug` [bigloo function]
`bigloo-compiler-debug-set!` *integer* [bigloo function]
`bigloo-debug` [bigloo function]
`bigloo-debug-set!` *integer* [bigloo function]
`bigloo-warning` [bigloo function]
`bigloo-warning-set!` *integer* [bigloo function]

These parameters control the debugging and warning level. The `bigloo-compiler-debug` is automatically controlled by the compiler command line `-g` option (see Chapter 13 [Command Line Parsing], page 153).

When a program is compiled in debug mode *lvl*, the compiler introduces a call to `(bigloo-debug-set! lvl)` before the evaluation of the first expression.

The `bigloo-debug` parameter is used to control traces (see Section 15.5 [Tracing], page 179).

`bigloo-trace` [bigloo function]
`bigloo-trace-set!` *list* [bigloo function]

Specifies the active trace (see `with-trace` form). The argument *list* is the list of symbols which are active and which triggers the display of a `with-trace` form.

`bigloo-trace-color` [bigloo function]
`bigloo-trace-color-set!` *bool* [bigloo function]

Enables/disables traces coloring (see Section 15.5 [Tracing], page 179).

`bigloo-trace-stack-depth` [bigloo function]
`bigloo-trace-stack-depth-set!` *integer* [bigloo function]

Controls the depth of the stack trace to be displayed on errors. With systems that supports shell variables (such as Unix) this parameter is dynamically adjusted according to the value of the SHELL variable *BIGLOOSTACKDEPTH*.

`bigloo-eval-strict-module` [bigloo function]
`bigloo-eval-strict-module-set!` *bool* [bigloo function]

When set to `#t` enables eval optimization that inlines operators calls. This optimization reduces the memory footprint of an application and it reduces the execution time.

`bigloo-dns-enable-cache` [bigloo function]
`bigloo-dns-enable-cache-set!` *bool* [bigloo function]

Enable/disable DNS name caching.

`bigloo-dns-cache-validity-timeout` [bigloo function]
`bigloo-dns-cache-validity-timeout-set!` *integer* [bigloo function]

Get/set the validity period for the DNS cache entries. It is expressed in seconds.

25 Explicit typing

Bigloo supports *type annotation* or *type information*. As shown in Section ref Module declaration, these annotations can be written both in the module clauses and in module bodies although module body type information is optional. It helps the compiler to produce better quality code and to reject incorrectly typed programs. Type annotations can describe both the result and formal parameter types for global functions and also types for local variable. Due to our module language design (in particular module initialization), Scheme global variables *cannot* support type information.

Types are either atomic types (see Section 26.1.6.1 [Atomic types], page 237), foreign types (see Section 26.1.6 [Defining an extern type], page 237), or a classes (see Section 9.1 [Class declaration], page 117).

Warning: All type annotations are ignored by the interpreter.

Module body type annotations are introduced by the following special forms.

`define` (*f*[*::type*] [*a*[*::type*]]...) *body* [bigloo syntax]
`define-inline` (*f*[*::type*] [*a*[*::type*]]...) *body* [bigloo syntax]
`let` ((*var*[*::type*] ...) ...) *body* [bigloo syntax]
`let` *loop* ((*var*[*::type*] ...) ...) *body* [bigloo syntax]
`let*` ((*var*[*::type*] ...) ...) *body* [bigloo syntax]
`letrec` ((*var*[*::type*] ...) ...) *body* [bigloo syntax]
`labels` ((*var*[*::type*] (*var*[*::type*]...) *b*) ...) *body* [bigloo syntax]

Type annotations are optional. That is, for any of these constructions, if a type annotation is missing, Bigloo uses the default generic type `obj` instead.

Here is an example of type annotated program:

```
(module example
   (export (vector-fill!::vector ::vector ::obj)))

(define (vector-fill! v filler)
  (let loop ((i::long (- (vector-length v) 1)))
     (if (< i 0)
         v
         (begin
            (vector-set! v i filler)
            (loop (- i 1))))))

(let ((v::vector (make-vector 3 4)))
   (vector-fill! v "dummy"))
```

The types that can be used in annotations are any of:

- the basic Scheme types `pair`, `null`, `bstring`, `bint` (presented in Section Section 26.1.6 [Defining an extern type], page 237).
- the basic extern types `long`, `int`, `char`, `string` presented in Section Section 26.1.6 [Defining an extern type], page 237.
- the compound extern types described in Section Section 26.1.6 [Defining an extern type], page 237.
- the types introduced by class declarations (Section Section 9.1 [Class declaration], page 117).

When a function that contains type annotation is exported, the type annotations must be written in the prototype of the function in the export clause. In that case the type annotation need to be written in the function definition:

```
(module foo
   (export (succ::int ::int)))

(define (succ x) (+ 1 x))
```

26 The C interface

We call all the pieces of program devoted to the interactions between Scheme and another language a *foreign interface*. In Bigloo, the foreign interface allows Scheme's functions and variables to be exported to a foreign language and foreign functions and variables to be imported into the Scheme code. Using the foreign interface requires two kind of operations.

- Declarations — type declarations, import declarations or export declarations.
- Foreign reference in the Scheme code.

Declarations take place in a special module clause, see Section 2.2 [Module Declaration], page 7, and reference to foreign variables within Scheme code requires no special construction. The current release of Bigloo includes a C and a Java interface. The Java connection is specified by the means of a `java` clause (see Chapter 27 [Java Interface], page 247). The C interface is active (that is the `extern` module clauses are read) only when compiling to C. So, when compiling to Jvm the binding declared in an `extern` clause are not bound.

Connecting Bigloo code with C is generally straightforward. To illustrate this simplicity, let us consider a simple example involving two source files. First a simple C file `sum.c` containing a single declaration:

```
int sum(int x, int y) { return x + y; }
```

Then, let us assume a Bigloo source code `main.scm` that makes uses of that C function:

```
(module foo
   (extern (sum::int (::int ::int) "sum"))
   (main main))

(define (main x)
   (print (sum (length x) 10)))
```

With a Unix installation of Bigloo, this program can be compiled and executed with the following commands:

```
$ gcc sum.c -c
$ bigloo main.scm sum.o -o main
$ ./main 1 2 3
```

The connection between Scheme and C is made particularly easy by Bigloo because the programmer is free from inserting conversion between Scheme values and C values. When needed, these are automatically inserted by the compiler.

26.1 The syntax of the foreign declarations

The syntax of *foreign* clauses is defined by:

```
<extern> ↦ <variable-clause>
     | <function-clause>
     | <include-clause>
     | <export-clause>
     | <type-clause>
```

Foreign clauses are automatically "transmitted" by the importation process. That is, if module `module1` imports a module `module2`, `module` treats the `extern` clauses of `module2` as though they were included in its own module declaration. Redefinition of a variable or a function already defined in an foreign clause is an error.

26.1.1 Automatic extern clauses generation

Extern clauses can be automatically generated using the Cigloo program which is distributed in the same package as Bigloo. Using Cigloo may be a good way to understand how C prototypes (and types) have to be declared in Bigloo. Cigloo reads C files and generates the Bigloo extern clauses for that files.

26.1.2 Importing an extern variable

The <variable-clause> denotes importation of variables.

```
<variable-clause> ↦ ( <typed-ident> <c-name>)
     | (macro <typed-ident> <string>)
     | (macro <typed-ident> (<typed-ident>+) <string>)
     | (infix macro <typed-ident> (<typed-ident>+) <string>)
```

Only extern "non-`macro`" variables are mutable (that is mutable using the `set!` construction). Bigloo does not emit "extern C prototype" for variables introduced by a `macro` clause. <string> is the C name of variable. The Scheme name of that variable is extracted from the <typed-ident>.

Here is an example of variable importations:

```
(module example
   (extern (c-var::double "c_var")
           (macro bufsiz::long "BUFSIZ")))

(print "c-var: " c-var)
(set! c-var (+ 1.0 c-var))
(print "c-var: " c-var)
(print "bufsize: " BUFSIZ)
```

26.1.3 Importing an extern function

Function are imported using the <function-clause>.

```
<function-clause> ↦ (<typed-ident> (<typed-ident>*) <string>)
     | (<typed-ident> (<typed-ident>+ . <typed-ident>) <string>)
     | (macro <typed-ident> (<typed-ident>*) <string>)
     | (macro <typed-ident> (<typed-ident>+ . <typed-ident>) <string>)
```

The function result type and Scheme name are extracted from the <typed-ident>; the <typed-ident> denotes the type of the function arguments and <string> is the C name of the function. Bigloo does not produce "C extern prototype" for macro functions (those introduced by `macro` clauses). If the typed identifier of the function does not contain any type information. Bigloo will emit a warning message when compiling and it will use a default C type (e.g. the `int` C type) as the return type of the function.

```
(module example
   (extern (macro prn::int (::string . ::long) "printf")))

(let ((n (read)))
   (prn #"fib(%d): %d\n" n (fib n)))
```

26.1.4 Including an extern file

C files can be included in the C code produced by using <include-clause>.

```
<include-clause> ↦ (include <string>)
```

26.1.5 Exporting a Scheme variable

A Scheme variable (or function) can be exported to the foreign world if and only if it is also exported using an `export` clause. Type information is given in the Scheme exportation, thus, the only requirement for a variable to be extern exported is to be given a foreign name. The foreign <export-clause> does this:

```
<export-clause> ↦ (export <ident> <string>)
```

Here is an example of exportation:

```
(module example
   (export (fib::long ::long))
   (extern (export fib "scheme_fib")))

(define (fib x) (if (< x 2) 1 ...))
```

26.1.6 Defining an extern type

New Bigloo types can be defined using extern <type-clause>. These newly introduced types can be used in any declaration (that is in any extern *or* Scheme module clause and in any Scheme variable or function definition). The syntax of <type-clause> is:

```
<type-clause> ↦ (type <ident> <type-def> <string>)
<type-def> ↦ <atomic-type>
     | <ident>
     | <struct-type>
     | <union-type>
     | <function-type>
     | <array-type>
     | <pointer-type>
     | <enum-type>
     | <opaque-type>
```

The symbol <ident> is the Scheme name of the introduced type and <string> is the C name of the type. When Bigloo produces the definition of a variable v of type s, it produces the following C code: s v;. This rules applies unless s is a pointer or an array and then, to produce a C definition, the name of the elements of the array or the elements pointed by the pointer type are used. Hence, if v is for instance `foo` and s is `(array int)` the produced C code will be: `int *foo`.

26.1.6.1 Atomic types

The atomic types are the pre-existing ones, defined in the standard Bigloo's library.

```
<atomic-type> ↦ <bigloo-type>
     | <c-type>
<bigloo-type> ↦ obj procedure
     | pair | nil | pair-nil
     | bint | blong | belong | bllong
     | bignum  |real | bbool | cnst
     | bstring | ucs2string | bchar | bucs2
     | vector | tvector | struct
     | tstruct | output-port | input-port
     | binary-port | unspecified | symbol | keyword
```

```
        | cell | date | process | exit
        | mutex | condvar | mmap
        | s8vector | u8vector | s16vector | u16vector
        | s32vector | u32vector | s64vector | u64vector
        | f32vector | f64vector
        | dynamic-env | opaque | foreign
    <c-type> ↦ cobj char
        | uchar | short
        | ushort | int | uint | long
        | ulong | slong | elong | llong
        | bool | string
        | file | double | float | void
        | function
```

The type `obj` denotes the super type of all Bigloo types (i.e., all Bigloo types, such as `procedure`, `pair`, ...) is an `obj`. The type `cobj` denotes the super of all C types (i.e., all preexisting C types such as `char`, `uchar`, `schar`, `short`, ...). The type `pair-nil` denotes values that are either pairs or the `()` value.

26.1.6.2 Struct and Union types

C struct and Union types can be declared in Bigloo using <struct-type> clauses:

```
<struct-type> ↦ (struct (<typed-ident> <string>)^+)
<union-type> ↦ (union (<typed-ident> <string>)^+)
```

This clause declared a C struct but C structure values *cannot* be handled by Bigloo. Instead Bigloo is able to handle *pointers to* C structure. Thus, in order to help the definition of extern types, when a struct named *struct* is defined, if it does not exists yet, Bigloo automatically defines a type *pointer to the structure*. This type is named *struct*`*`.

When a pointer to a structure type is defined, Bigloo automatically produces functions to manipulate objects of this type. Let us suppose the type definition of *struct*`*`:

```
(type struct
   (struct (id1::type1 name1)
           ...
           (idn::typen namen))
```

The following functions are created:

- A creator:

  ```
  (struct*::struct* ::type_1 ... ::type_n)
  ```

  This function allocates a fresh *struct*`*` (in the same heap as any Scheme value) and fills the fields of the C structure with the proper values provided in the call.

- A type checker:

  ```
  (struct*?::bool obj::obj)
  ```

  This function returns `#t` if and only if the argument *obj* is of type *struct*`*`.

- A null checker:

  ```
  (struct*-null?::bool ::struct*)
  ```

  This function returns `#t` if and only if its argument is `Null`.

- A null creator:

  ```
  (make-null-struct::struct*)
  ```

This function creates a `NULL` value of type *struct**.

- An equality checker:

```
(=struct*?::bool ::struct* ::struct*)
```

This function returns `#t` if and only if its arguments are equal.

- Accessors and mutators:

```
(struct*-id_1::type_1 ::struct*)
(struct*-id_1-set!::obj ::struct* ::type_1)
...
```

These functions read and store field values.

Here is an example of structure usage:

```
(module foo
   (extern
     (include "named_point_declaration.h")
     (type named-point
           (struct (x::double "x")
                   (y::double "y")
                   (name::string "name"))
           "struct named_point")
     (c-print-point::int (named-point*) "ppoint")))

(define (scheme-print-point point)
   (print "point*-name: " point
          " x: " (named-point*-x point)
          " y: " (named-point*-y point)))

(let ((orig (named-point* 0.0 0.0 "orig")))
   (if (named-point*-null? orig)
       (error "bigloo" "cannot allocate point" orig)
       (begin
          (c-print-point orig)
          (scheme-print-point orig))))
```

26.1.6.3 C pointers

C pointers are defined by the <pointer-type>

```
<pointer-type> ↦ (pointer <ident>)
```

<ident> is the name of a previously defined type. Let us suppose the pointer type declaration:

```
(type ptr (pointer ident) ...)
```

If *ident* is the name of a structure type, Bigloo automatically creates structure accessors (see Section 26.1.6.2 [C structures and unions], page 238). Otherwise, it creates the following functions:

- A creator:

```
(make-ptr::ptr nb::long)
```

This function allocates memory for *nb* elements of type *ident* and returns a *ptr* to this zone. The memory is filled with the C `Null` value.

- A type checker:

```
(ptr?::bool obj::obj)
```

This function returns `#t` the argument *obj* is of type *ptr* and `#f` otherwise.

- A null checker:

  ```
  (ptr-null?::bool ::ptr)
  ```

  This function returns `#t` if its argument is `Null` and `#f` otherwise.

- A null creator:

  ```
  (make-null-ptr::ptr*)
  ```

  This function creates a `NULL` value of type *ptr**.

- An equality checker:

  ```
  (=ptr*?::bool ::ptr* ::ptr*)
  ```

  This function returns `#t` if its arguments are equal and `#f` otherwise.

- Accessors and mutators:

  ```
  (ptr-ref::ident ::ptr ::long)
  (ptr-set!::obj ::ptr ::long ::ident)
  ```

  These functions read and store field values.

Here is an example of a program using pointer types:

```
(module foo
   (extern
    (type double* (pointer double) "double *")))

(define (make-vect::double* x y z)
   (let ((vect (make-double* 3)))
      (double*-set! vect 0 x)
      (double*-set! vect 1 y)
      (double*-set! vect 2 z)
      vect))

(define (vect-norm vect::double*)
   (sqrt (+ (expt (double*-ref vect 0) 2)
    (expt (double*-ref vect 1) 2)
    (expt (double*-ref vect 2) 2))))

(print (vect-norm (make-vect 1.2 4.5 -4.5)))
```

26.1.6.4 C null pointers

It may be convenient to build C null pointers. Several means can be used. In particular, foreign structures and pointers are provided with Null creators. For other foreign types, the easiest one is likely to be a `pragma` form. For instance, in order to create a null pointer to a `double*` type, one may use:

```
(pragma::double* "((double *)0L)")
```

`string-ptr-null?` *string* [bigloo procedure]
`void*-null?` *void** [bigloo procedure]

These two predicates checks if there argument is the C `NULL` value.

`make-string-ptr-null` [bigloo procedure]
`make-void*-null` [bigloo procedure]

These two constructors creates *null* foreign values.

26.1.6.5 C arrays

C arrays are defined by the <array-type>

```
<array-type> ↦ (array <ident>)
```

<ident> is the name of a previously defined type. Array types are similar to pointer types except that they include their size in their type definition string. Let us suppose the array type declaration:

```
(type array (array ident) ...)
```

If *ident* is the name of a structure type, Bigloo automatically creates structures accessors (see Section 26.1.6.2 [C structures and unions], page 238). Otherwise, it creates the following functions:

- A creator:

  ```
  (make-array::array)
  ```

  This function allocates memory for the array *array*. The memory is filled with the C `Null` value.

- A type checker:

  ```
  (array?::bool obj::obj)
  ```

  This function returns `#t` if the argument *obj* is of type *array* and `#f` otherwise.

- A null checker:

  ```
  (null-array?::bool ::array)
  ```

  This function returns `#t` if the argument *obj* is `Null` and `#f` otherwise.

- An equality checker:

  ```
  (=array*?::bool ::array* ::array*)
  ```

  This function returns `#t` if its arguments are equal and `#f` otherwise.

- Accessors and mutators:

  ```
  (array-ref::ident ::array ::long)
  (array-set!::obj ::array ::long ::ident)
  ```

  These functions read and store field values.

Here is an example of a program using array types:

```
(module foo
   (extern
    (type double* (array double) "double [ 10 ]")))

(define (make-vect::double* x y z)
   (let ((vect (make-double*)))
      (double*-set! vect 0 x)
      (double*-set! vect 1 y)
      (double*-set! vect 2 z)
      vect))

(define (vect-norm vect::double*)
   (sqrt (+ (expt (double*-ref vect 0) 2)
    (expt (double*-ref vect 1) 2)
    (expt (double*-ref vect 2) 2))))


(print (vect-norm (make-vect 1.2 4.5 -4.5)))
```

26.1.6.6 C functions

C function types are introduced by the <function-type> clause:

```
<function-type> ↦ (function <ident> (<ident>*))
```

Let us suppose the array type declaration:

```
(type fun (function res (arg*)) ...)
```

Bigloo creates the following functions:

- A type checker:

  ```
  (fun?::bool obj::obj)
  ```

  This function returns `#t` if the argument *obj* is of type *fun* and `#f` otherwise.

- An equality checker:

  ```
  (=fun*?::bool ::fun* ::fun*)
  ```

  This function returns `#t` if and only if its arguments are equal.

- Caller:

  ```
  (fun-call::res f::fun a::ta ...)
  ```

  This function invokes f with the arguments $a \ldots an$.

Suppose we have to use in Scheme the following C variable:

```
double (*convert)(char *);
```

It can be done as in:

```
(module foo
   (extern
    (type *string->double
          (function double (string))
                    "double (*)(char *)")
    (macro cv::*string->double "convert")))

(print (*string->double-call cv "3.14"))
```

26.1.6.7 C enums

This form defines `enum` types.

```
<enum-type> ↦ (enum (<ident> <string>)... ...)
```

Let us suppose the type:

```
(type enum
   (enum (id_1 name_1)
         ...
         (id_n name_n)))
```

Bigloo creates the following functions:

- Creators:

  ```
  (enum-id_1::enum)
  ...
  (enum-id_n::enum)
  ```

  These functions create *enum* values.

- A type checker:

  ```
  (enum?::bool obj::obj)
  ```

  This function returns `#t` if the argument *obj* is of type *enum* and `#f` otherwise.

- An equality checker:
  `(=enum?::bool ::enum ::enum)`

  This function returns `#t` if the arguments are equal and `#f` otherwise.

Here is an example of Scheme code using *enum* type.

```
(module foo
   (extern
    (type gizmo
          (enum (titi "titi")
                (tutu "tutu")
                (tata "tata"))
          "enum toto")))

(let ((v1 (gizmo-titi))
      (v2 (gizmo-tutu)))
   (print (=gizmo? v1 v2)))
```

26.1.6.8 C opaques

This form defines `opaque` types.

`<opaque-type>` ↦ `(opaque)`

Let us suppose the type:

`(type opa (opaque) ...)`

Bigloo creates the following functions:

- A type checker:
  `(opa?::bool obj::obj)`

  This function returns `#t` if the argument *obj* is of type *opa* and `#f` otherwise.
- An equality checker:
  `(=opa?::bool ::opa ::opa)`

  This function returns `#t` if the arguments are equal and `#f` otherwise.

Opaque types are relevant when a C value must transit via a Scheme function from a C function to another C function. The value can't be used in Scheme because no accessors are defined over that type it can only be send back to a C function.

Here is an example of Scheme code using *opaque* type.

```
(module foo
   (extern (type filedes (opaque) "FILE *")
           (macro _fopen::filedes (::string ::string) "fopen")
           (_fgetc::int (::filedes) "fgetc")
           (_fclose (::filedes) "fclose"))
   (export (fopen::filedes ::bstring ::bstring)
           (fclose ::filedes)
           (fgetc::char ::filedes)))

(define (fopen fname mode)
   (_fopen fname mode))

(define (fclose filedes)
   (_fclose filedes))

(define (fgetc filedes)
   (integer->char (_fgetc filedes)))
```

Note: To illustrate the default type compilation of extern function, we have voluntary introduced an incomplete declaration for the `fclose` function. This will make Bigloo to produce a warning when compiling that module.

26.2 The very dangerous "pragma" Bigloo special forms

Bigloo has a special form which allows the inclusion of C text into the produced code. It is *only* applicable to the C back-end. In particular, the JVM back-end (see Chapter Chapter 27 [Java Interface], page 247) does not support it.

`pragma::`*`ident`* *string* [*args*] [bigloo syntax]
`free-pragma::`*`ident`* *string* [*args*] [bigloo syntax]
`pragma::`*`ident ident`* [bigloo syntax]

This force Bigloo to include *string* in the produced C code as a regular C fragment of code. This form must not be used without an in depth understanding of Bigloo C code production; with unskilled use, the produced C file may be unacceptable to the C compiler.

Values can be passed to a `pragma` form, being referenced in *string* by expressions of the form `$`*`number`*. Such expression are replaced by the corresponding values, the number of referenced values in *string* being exactly the number of values provided. Here is an example of `pragma` usage:

```
(define (fibo x::long)
   (pragma "printf( \"fib(%d):%d\\n\", $1, $2 );"
           x
           (fib x)))
```

Arguments provided to a pragma form are not converted during compilation. Hence, pragma arguments can be of any types, including, foreign types.

A pragma result type can be specified using the notation `pragma::`*`name`* where the default type is `unspecified`. Then, for instance, the expression `(pragma::bool "$1 == 0" x)` will be considered to be returning a object of type `bool` (C boolean) while the expression `(pragma "$1 == 0" x)` will be considered by Bigloo to be returning the `unspecified` typed object.

The compiler assumes that a `pragma` forms operates a side effects and that it writes into its parameters. This assumption no long holds with `free-pragma`. This is the only difference between the two forms.

The last form `pragma` *`ident`* enables "injecting" a Scheme mangled identifier into the generated C code. Example:

```
(let ((x/y 3))
  (pragma "$1 = BINT( 24 )" (pragma x/y))
  (print x/y))
```

26.3 Name mangling

In order to avoid name clashes, Bigloo uses name mangling when compiling to C or to Jvm. The name mangling for a Scheme identifier may be overridden by the means of an extern `export` clause (see Section Section 26.1.5 [Exporting a Scheme variable], page 237).

Four public functions may be used to mangle and to demangle Scheme identifiers:

`bigloo-mangle` *string* [bigloo procedure]
Mangle the identifier *string*.

`bigloo-module-mangle` *string1 string2* [bigloo procedure]
Mangle the identifier *string1* that belongs to module *string2*.

`bigloo-mangled?` *string* [bigloo procedure]
Returns `#t` if *string* has been computed by the `bigloo-mangle` or `bigloo-module-mangle` function.

`bigloo-class-mangled?` *string* [bigloo procedure]
Returns `#t` if *string* is a mangled name of a Bigloo class.

`bigloo-need-mangling` *string* [bigloo procedure]
Returns `#t` if *string* requires name mangling because it is not a C or Jvm valid identifier.

`bigloo-demangle` *string* [bigloo procedure]
Demangle previously mangled identifiers:

```
(let ((id "foo!")
      (module "a-module"))
   (let ((mangled (bigloo-module-mangle id module)))
      (multiple-value-bind (new-id new-module)
         (bigloo-demangle mangled)
         (and (string=? id new-id) (string=? module new-module)))))
   ⇒ #t
```

`bigloo-class-demangle` *string* [bigloo procedure]
Demangle previously mangled class identifier.

26.4 Embedded Bigloo applications

It is possible to design and realize embedded Bigloo applications. This facility is useful for adding a new Scheme part to an already existing C program. The C part of the program has only to enter the Bigloo initialization, hence, it can call any Bigloo function.

Normally, Bigloo creates an initialization function called `main` when it reads a `main` module clause. To use an embedded Bigloo program, such an initialization function would have to be created but with a different name. Changing the name can be be done using the following Bigloo option: `-copt "-DBIGLOO_MAIN=<new-name>"`. To prevent exit from the program after `<new-name>` is executed, the following Bigloo option must be used: `-copt "-DBIGLOO_EXIT='BUNSPEC,'"`.

A very important part of designing embedded Bigloo programs is being sure that all used Bigloo modules are correctly initialized and the normal way to initialize them is to use `with` clauses in the module which contains the `main` clause.

An example of an embedded program can be found in the distribution's examples directory.

26.5 Using C bindings within the interpreter

To be able to get access to foreign functions within the Bigloo interpreter, some extra measurements have to be taken. The foreign functions have to be present in the interpreter binary, which means you have to compile a custom interpreter. Fortunately, this is easy. What has to be done is to wrap the foreign functions within Scheme and make an interpreter module.

Let us consider an example where a C function `get_system_time` returning and `int` is used in an interpreter. (When linking, be sure to add the `.o` file containing the `get_system_time`.)

The `ffi-interpreter.scm` file:

```
(module ExtendendInterpreter
   (import (wrapper "wrapper.scm"))
   (main main))

(define (main argv)
   (repl))
```

The `wrapper.scm` file:

```
(module wrapper
   (extern (macro %get-system-time::int () "get_system_time"))
   (export (get-system-time))
   (eval (export-exports))

(define (get-system-time)
  (%get-system-time))
```

Compile and link your application with something like:

```
cc gettime.c -c gettime.o
bigloo wrapper.scm -c
bigloo ffi-interpreter.scm wrapper.o gettime.o
```

27 The Java interface

When the Bigloo is configured for a JVM back-end support, the compiler is able to produce Java class file instead of C files. In order to produce JVM class files, use the `-jvm` compiler option. Example:

```
$ cat > foo.scm
(module foo (main main))
(define (main argv)
  (print "Hello world: " argv))
$ bigloo -jvm foo.scm
$ a.out
  ⊣ Hello world: (a.out)
```

27.1 Compiling with the JVM back-end

27.1.1 Compiler JVM options

All the compiler options that control the compilation (optimization options, debugging options, etc.), can be used in conjunction with the `-jvm` option. However, the `-jvm` option *MUST* be the first compiler option on the command line.

In order to prevent the compiler to produce a script shell file to run the program, it is required to use simultaneously the `-jvm` and `-c` options.

27.1.2 Compiling multi-modules applications

In order to compile and link multi-modules applications, it is required to specify the association between Scheme source modules and Java qualified type names. This task is generally complex because of the annoying mapping that exists from Java class names and the operating file system names. In order to get rid of this problem, the Bigloo standard distribution contains a tool, `jfile`, that automatically produces Bigloo Module/Java classes association files. The default name for such a table is `.jfile`. When compiling a module, Bigloo checks if a `.jfile` exists in the current directory, if it exists, the file is read. The compilation option `-jfile` may be used to specify an alternative jfile name. Example:

```
$ cat > foo.scm
(module foo (export (foo))) (define (foo) 'foo)
$ cat > bar.scm
(module bar (export (bar))) (define (bar) 'bar)
$ cat > hux.scm
(module hux (export (hux))) (define (hux) 'hux)
$ cat > main.scm
(module main (main main) (import foo bar hux)
(define (main argv)
  (print (foo))
  (print (bar))
  (print (fhux)))
$ afile *.scm > .afile
$ jfile *.scm > .jfile
```

```
$ bigloo -jvm -c foo.scm
$ bigloo -jvm -c bar.scm
$ bigloo -jvm -c hux.scm
$ bigloo -jvm main.scm foo.class bar.class hux.class
```

For an explanation about the `.afile`, see Chapter 2 [Modules], page 7.

27.2 JVM back-end and SRFI-0

The currently running back-end may be tested by the means of the SRFI-0 `cond-expand` form (see Chapter 30 [SRFIs], page 269). That is, when the JVM is ran, the `bigloo-jvm` clause is true. Otherwise, the `bigloo-c` is true. Example:

```
$ cat > foo.scm
(module foo (main main))
(define (main argv)
  (cond-expand
    (bigloo-jvm (print "JVM back-end"))
    (bigloo-c (print "C back-end"))
    (else (error "main" "unsupported back-end" #unspecified))))
$ bigloo -jvm foo.scm
$ a.out
 ⊣ JVM back-end
$ bigloo foo.scm
$ a.out
 ⊣ C back-end
```

27.3 Limitation of the JVM back-end

The JVM back-end supports the entire Bigloo source language but the `call/cc` function. More precisely, using the JVM back-end, the continuation reified in a `call/cc` form can only be invoked in the dynamic extent of that form.

The other restrictions of the C back-end apply to the JVM back-end. Mainly,

- Bigloo is not able to compile all the tail recursive call without stack consumption (however, most of the tail recursive calls are optimized by Bigloo and don't use stack activation frames).
- Bigloo compiled applications do not check for arithmetic overflow.
- When compiling to Jvm, the `extern` module clauses are not used.
- Jvm runtime system does support the following function `chdir`.
- Jvm runtime system support for `chmod` is restricted.
- In order to read a shell variable from a Bigloo compiled Jvm program, you have to use the Bigloo link option `-jvm-env` *when* linking that program. However, some shell variables are automatically defined (`HOME`, `USER`, `CLASSPATH` and `TMPDIR`.
- JVM code generation does not support `pragma` forms.

27.4 Connecting Scheme and Java code

When compiling and linking with the JVM back-end, Bigloo source code may use the Java API. That is, Bigloo Scheme source code may use (refer or set) Java static variables, Bigloo source code may call static or virtual Java methods. In addition, Bigloo variables and functions may be exported to Java, that is use, set or called in Java source code. Java module clauses are enabled (read and parsed) only when compiling to JVM byte code.

Java definitions are declared in Bigloo modules by the mean of a Bigloo module clause: the *java* module clause. The syntax of a *Java* clause is defined by:

```
<java> ↦ <declare-class-clause>
     | <declare-abstract-class-clause>
     | <extend-class-clause>
     | <array-clause>
     | <export-clause>
```

As for the *extern* clause, *java* clauses are automatically "transmitted" by the importation process. That is, if module `module1` imports a module `module2`, `module` treats the `java` clauses of `module2` as though they were included in its own module declaration. Redefinition of a variable or a function already defined in an java clause is an error. However, the definition of a Java class or an Java abstract class may be enriched from module to module.

27.4.1 Automatic Java clauses generation

Java clauses can be automatically generated using the Jigloo program which is distributed in the same package as Bigloo. Using Jigloo may be a good way to understand how Java classes, methods, and variables have to be declared in Bigloo. Jigloo reads Java *class* files and generate the Bigloo java clauses for that classes.

27.4.2 Declaring Java classes

The <declare-class-clause> clause denotes importation of Java classes.

```
<declare-class-clause> ↦ (class <typed-ident> <slot>* <string>)
<slot> ↦ <field> | <method> | <constructor>
<field> ↦ (field <modifier> <typed-ident> <string>)
<method> ↦ (method <modifier> <typed-ident> (<typed-ident>*) <string>)
<constructor> ↦ (constructor <ident> (<typed-ident>*))
<modifier> ↦ public | private | protected
     | static | final | synchronized | abstract
```

When the compiler encounters a Java class declaration, it automatically creates a predicate. If the class identifier is `id`, the predicate is named `id?`. In addition, the compiler generates functions that fetch and set the field values. For a field named `f`, these functions are named `id-f` and `id-f-set!`. Methods and constructors are also always prefixed the name of the class. That is, for a method named `m` of a class `k`, the Scheme name of the method is `k-m`.

Example:

```
(module java-example
   (java (class point
            (constructor new-default ())
            (field x::int "x")
```

```
            (method show::void (::point) "show")
            (method static statistics::int () "PointStatistics")
            "Point")
         (class point-3d::point
            "Point3D")))

(let ((p (point-new-default)))
   (print (point? p))   ⊣ #t
   (point-x-set! p 3)
   (print (point-x p))) ⊣ 3
```

27.4.3 Declaring abstract Java classes

A Bigloo abstract Java class declaration corresponds to a Java interface. It cannot be instantiate but regular classes may inherit from it.

```
<declare-abstract-class-clause> ↦ (abstract-class <typed-ident> <slot>* <string>)
```

27.4.4 Extending Java classes

A class definition may be split into several pieces. One class declaration (see <declare-class-clause>) and several extensions. The syntax for a Java class extension is:

```
<extend-class-clause> ↦ (class <typed-ident> <slot>*)
```

Example:

```
(module java-example2
   (import java-example)
   (java (class point
            (field y::int "y")
            (field static num::int "point_num")
            (constructor new (::int ::int)))))
```

27.4.5 Declaring Java arrays

Java arrays may be allocated and used inside Scheme code. The syntax of a Java array module clause is:

```
<array-clause> ↦ (array <ident> <typed-ident>)
```

The <typed-ident> must refer to the name of an existing type (i.e., a primitive Bigloo type, a Bigloo class, an already defined Java class or an already defined Java array). For an array named `ar`, Bigloo generates:

- a creator named `make-ar` which is a function of one integer argument.
- a predicate named `ar?`.
- a getter named `ar-ref` which is a function of two arguments: an array and one integer argument.
- a setter named `ar-set!` which is a function of three arguments: an array, an integer, and a value of the array item types.
- a length named `ar-length`.

Example:

```
(module foo
   (java (array int* ::int)
         (class bar
            (method static hello::int (::int*) "hello")
```

```
            "bar"))
   (main main))

(define (main argv)
   (let ((tab (make-int* 2)))
      (int*-set! tab 0 3)
      (int*-set! tab 1 6)
      (print (bar-hello tab))))
```

27.4.6 Exporting Scheme variables

As for the C connection, a Scheme variable (or function) can be exported to the Java world if and only if it is also exported using an `export` Java clause. Type information is given in the Scheme exportation, thus, the only requirement for a variable to be Java exported is to be given a Java name. The Java <export-clause> does this:

```
<export-clause> ↦ (export <ident> <string>)
```

Here is an example of exportation:

```
(module example
   (export (fib::long ::long))
   (java (export fib "scheme_fib")))

(define (fib x) (if (< x 2) 1 ...))
```

Here is a concrete example that illustrates how to invoke Scheme functions from Java code. First, here is a Java source file:

```
// arr.java
import java.io.*;
import bigloo.*;

public class arr {
   public static int hello( int[] tab ) {
      System.out.println( "tab[ 0 ]: " + tab[ 0 ] );
      System.out.println( "tab[ 1 ]: " + tab[ 1 ] );
      System.out.println( "<callback: " + main.callback( 10 ) + ">" );
      System.out.println("tab = main.squareJavaArray(tab);");
      tab = main.squareJavaArray(tab);
      System.out.println( "tab[ 0 ]: " + tab[ 0 ] );
      System.out.println( "tab[ 1 ]: " + tab[ 1 ] );
      return tab.length;
   }
}
```

Then the main Scheme file

```
;; main.scm
(module main
   (java (array int* ::int)

      (class bar
 (method static hello::int (::int*) "hello")
 "bar"))

   (export (callback::int ::int))
   (export (squareJavaArray::int* ::int*))

   (main main))

(define (main argv)
```

```
  (let ((tab (make-int* 2)))
    (print "tab length: " (int*-length tab))
    (int*-set! tab 0 3)
    (int*-set! tab 1 6)
    (display (int*-ref tab 1))
    (newline)

    (display "(java-array-int->scheme-vector tab) : ")
    (display (java-array-int->scheme-vector tab))
    (newline)
    (display "(square-vector (java-array-int->scheme-vector tab)) : ")
    (display (square-vector (java-array-int->scheme-vector tab)))
    (newline)

    (print (bar-hello tab))))

(define (callback x)
   (+ 1 x))

(define (square x)
   (* x x))

(define (square-list L)
   (map square L))

(define (square-vector V)
   (vector-map square V))

(define (sum-vector V)
   (apply + (vector->list V)))

(define (squareJavaArray A)
   (scheme-vector->java-array-int
      (square-vector (java-array-int->scheme-vector A))))

(define (java-array-int->scheme-vector A)
   (let* ((len (int*-length A))
  (res (make-vector len)))
      (display len)
      (newline)
      (let loop ((i 0))
 (vector-set! res i (int*-ref A i))
 (if (= i (- len 1))
     res
     (loop (+ i 1))))))

(define (scheme-vector->java-array-int V)
   (let* ((len (vector-length V))
  (res (make-int* len)))
      (let loop ((i 0))
 (int*-set! res i (vector-ref V i))
 (if (= i (- len 1))
     res
     (loop (+ i 1))))))
```

To compile this example:

```
$ bigloo -jvm main.scm -c
$ javac arr.java -classpath=bigloo.zip:.
```

```
$ bigloo -jvm main.scm arr.class
```

To run it:

```
$ ./a.out
tab length: 2
6
(java-array-int->scheme-vector tab) : 2
#(3 6)
(square-vector (java-array-int->scheme-vector tab)) : 2
#(9 36)
tab[ 0 ]: 3
tab[ 1 ]: 6
<callback: 11>
tab = main.squareJavaArray(tab);
2
tab[ 0 ]: 9
tab[ 1 ]: 36
2
```

27.4.7 Bigloo module initialization

By default Bigloo modules are initialized when the application starts. It might be convenient to initialize the module when the Java classes implementing the Bigloo modules are loaded. It is possible to drive the Bigloo compiler to introduce code inside the Java class constructors for initializing the modules. This is the role of the `-jvm-cinit-module` compiler option.

27.5 Performance of the JVM back-end

We are currently improving and investigating about the performance of the JVM back-end. JVM performance is extremely sensitive to the host platform (for instance, very unfortunately, Linux seems to be a poor platform to run JVM code). Currently, it seems that the JVM back-end produces codes that are in between 4 times and 10 times slower than codes produced by the C back-end. The ratio between JVM and C is subject to changes. The gap between JVM and C code is bound to bridge because of the huge amount of efforts applied to efficiently implement Java virtual machines.

28 Bigloo Libraries

Bigloo libraries are collections of global bindings (global variables and global functions). Bigloo libraries are built on the top of the host operating system (e.g. Unix) libraries. Because Bigloo uses modules, a library is not only a bundle of compiled codes and memory locations. A Bigloo library is split into several files:

- one *heap* that describes the variables and functions of the library.
- several host library files (safe and unsafe versions of the compilation and also *eval libraries* that contain the code that binds the variables and functions to the evaluator).
- possibly, C header files.
- possibly, an initialization file.

Let's consider, for example, a library that implements the `format` Common Lisp facility. Let's suppose we name this library `bformat` and that the library number is `1.0`. Using a Unix machine, the Bigloo library will consist of the following files:

- `bformat.heap`: the heap file.
- `bformat.init`: the initialization file.
- `libbformat_s-1.0.a`, `libbformat_s-1.0.so`, `libbformat_u-1.0.a`, `libbformat_u-1.0.so`, `libbformat_eu-1.0.so`, and `libbformat_es-1.0.so`:: the Unix library files. The file names with a `_u` are libraries compiled in *unsafe* and *optimized* mode. By convention the library using the `_s` suffix are *safe* libraries, `_p` are profiling libraries, `_d` debug libraries, `_es` and `_eu` eval libraries.
- `bformat.h`: an include file.

28.1 Compiling and linking with a library

From the user standpoint, using a library can be made two ways:

- Using the Bigloo `-library` *`lib-name`* option where *lib-name* is the name of the Bigloo library (not the name of one of the Unix files implementing the library). The name of the library must be *lower case*. For instance:

  $ bigloo foo.scm -library bformat

- Using the module clause `library`. This second solution avoids using a special compilation option. For instance, this module will automatically compile and link with the `bformat` library:

```
(module foo
   (library bformat))

...
(format ...)
```

When a Bigloo library *`lib`* is used, Bigloo automatically searches for a file called *`lib`*`.init` (the "init file"). If such a file exits, it is loaded at *compile-time*. For instance, the init file may be used to specify compilation flags or to define macros used by the compiler. The initialization file may affect any of the global parameters of the Bigloo compiler. For instance, a Bigloo library supporting SSL connections would likely need

a native library. Setting the compiler variable `*ld-post-options*` has this effect. For instance, one may define an initialization file such as:

```
(cond-expand
   (bigloo-compile
    (set! *ld-post-options* (string-append "-lssl " *ld-post-options*)))
   (bigloo-eval
    #unspecified))
```

When a Bigloo library `lib` is used, the Bigloo linker automatically looks at a library to be linked against the application. The name of the file containing the library depends on the operating system and the back-end used. For instance, under Unix, for a library called *NAME*, the Bigloo linker searches for a file called `lib`*`NAME`*`_[s|u]-`*`VERSION`*`.a` or `lib`*`NAME`*`_[s|u]-`*`VERSION.DYNLIB-SUFFIX`* in the compilation linker path when using the native back-end. It searches for a file *`NAME`*`_[s|u]-`*`VERSION`*`.zip` when the JVM back-end is used.

This default *NAME* can be overridden in the initialization file. The function `declare-library!` associates a Bigloo library name and a system name.

`declare-library!` *ident* [*attributes*] [library procedure]

All the attributes are optional.

- `version:` the version number of the library. This defaults to the Bigloo version number.
- `basename:` the base of the filename containing the library. This defaults to the library name.
- `srfi:` a list of symbols denoting the SRFI 0 features implemented by this library. Registered SRFIs may be tested by the `cond-expand` form (see Chapter 30 [SRFIs], page 269). This defaults to an empty list.
- `dlopen-init:` a function to be invoked when the library is dynamically loaded using the function `dynamic-load`. This defaults to `#f`.
- `module-init:` a module to be initialized when the library is loaded. This defaults to `#f`.
- `eval-init:` a module to be initialized for binding the library exports in the interpreter. This defaults to `#f`.
- `class-init:` the JVM or .NET class name containing the module to be initialized. This defaults to `#f`.
- `eval-init:` the JVM or .NET class name containing the module to be initialized for eval. This defaults to `#f`.
- `init:` a function to be invoked when a library is loaded. This defaults to `#f`.
- `eval:` a function to be invoked when a library is loaded by the interpreter. This defaults to `#f`.
- `eval:` a function to be invoked when a library is loaded by the interpreter. This defaults to `#f`.

Examples:

- The following declares a library named `foo`. When loaded, the Bigloo runtime system will seek file named `libfoo_s-3.4a.so`, `libfoo_u-3.4a.so`, `libfoo_es-3.4a.so`, and `libfoo_eu-3.4a.so`.

```
(declare-library! 'foo)
```

- The following declares a library named `pthread`. When loaded, the Bigloo runtime system will seek a file named `libbigloopth_s-1.1a.so`, `libbigloopth_u-1.1a.so`, `libbigloopth_es-1.1a.so`, `libbigloopth_eu-1.1a.so`. Once the library loaded, the SRFI-0 features `pthread` and `srfi-18` will be bound. When loading the library, the two modules `__pth_thread` and `__pth_makelib` will be initialized. In the JVM version these modules are compiled in the classes `"bigloo.pthread.pthread"` and `"bigloo.pthread.make_lib"`.

```
(declare-library! 'pthread
                  :basename "bigloopth"
                  :version "1.1a"
                  :srfi '(pthread srfi-18)
                  :module-init '__pth_thread
                  :module-eval '__pth_makelib
                  :class-init "bigloo.pthread.pthread"
  :class-eval "bigloo.pthread.make_lib")
```

`library-translation-table-add!` *ident name* [library procedure]
`library-translation-table-add!` *ident name version* [library procedure]
`library-translation-table-add!` *ident name version* [library procedure]
:dlopen-init initsym

The function `library-translation-table-add!` is obsolete. It should no longer be used in new code. It is totally subsumed by `declare-library!`. The function `library-translation-table-add!` is still documented for enabling readers to understand old Bigloo source code.

This function registers a *name* for the library *id*. An optional *version* can be specified. The optional named argument `dlopen-init` gives the base name of the initialization entry point of a library.

Imagine that we would like to name our `bformat` library `bigloobformat`. This can be achieved by adding the following expression in the initialization file.

```
(library-translation-table-add! 'bformat "bigloobformat")
```

Using this translation, on a Unix platform, the library used during the linking will be named: `libbigloobformat_s-<BIGLOO-VERSION>.a`. In order to change the `<BIGLOO-VERSION>` to another suffix, such as `1.0`, one may use:

```
(library-translation-table-add! 'bformat "bigloobformat" "1.0")
```

In such a case, the library searched will be named `libbigloobformat_s-1.0.a`.

Specifying a `#f` prevents the insertion of any suffix. Hence,

```
(library-translation-table-add! 'bformat "bigloobformat" #f)
```

instructs the compiler to look at a library named `libbigloobformat_s.a`.

28.2 Library and inline functions

It is illegal for libraries to include inline functions that make use of new foreign types. By "new foreign type", we mean foreign types that are defined inside the library. A library may contain inline functions but these inline functions must not call functions using foreign types in their prototypes. Including inline functions making use of foreign C types will make the compiler fail when compiling user code, prompting type errors. A library may contains non-inline functions that make use of new foreign types.

28.3 library and eval

The function `library-load` loads a library in the interpreter.

`library-exists?` *ident . path* [library procedure]
Checks if the library *ident* exists for the current back-end.

The regular Bigloo library paths are scanned unless optional *path*s are sent to the function.

`bigloo-library-path` [library procedure]
`bigloo-library-path-set!` [library procedure]
These functions get and set the default path (a list of strings) for loading libraries.

`library-load` *ident . path* [library procedure]
Loads a library in the interpreter. In addition to dynamically loading the library, this function tries to load the `_es` version of the library if it is linked against the safe Bigloo library version or the `_eu` version if it is linked against the unsafe version of the Bigloo library.

Searches for libraries occur in the regular Bigloo library paths unless optional *path*s are sent to the function.

This version may be used for automatically exporting bindings to the interpreter. In general, the `_es` and `_eu` libraries are simple libraries that contain only one module, the module that is used to build the heap-file. For instance, let's consider an implementation of a library for SSL programming. This library is composed of a single implementation module `__ssl_ssl`. The library is build using a heap file:

```
(module __ssl_makelib
   (import __ssl_ssl))
```

Changing this file for:

```
(module __ssl_makelib
   (import __ssl_ssl)
   (eval   (export-all)))
```

enables the construction of the `_es` and `_eu` libraries.

When the system loads a dynamic library, it *initializes* it. For that it expects to find *initialization entry points* in the dynamic libraries that are named after the library's name. More precisely, for the `LIB_s` library, the loader seeks the entry point named `"LIB_s"` and for the `LIB_es`, it seeks `"LIB_es"`. The name of the initialization entry of a library can be changed using the `declare-library!` function. If that named is changed, one module of the library must contain an `option` module clause that sets the variable `*dlopen-init*` with the name of the initialization entry point.

Since Bigloo 3.1a, the runtime system supports a better way for initializing libraries. *Initialization* modules can be associated with a library. When loaded, these modules are automatically initialized. This new method fits harmoniously with the Bigloo initialization process and it relieves users from any requirement to annotate the source code of the library.

For instance, if a library initialization file contains the following declaration:

```
(declare-library! 'foo :module-init 'foo)
```

Then, the library must implement the `foo` module.

```
(module foo
  (import ...)
  ...)
```

In addition if the library binds variables, functions, or classes in the interpreter then, an `eval-init` clause must be added to the class declaration:

```
(declare-library! 'foo :module-init 'foo :eval-init 'foo-eval)
```

Then, the module `foo-eval` must be implemented in the `libfoo_es` and `libfoo_eu` libraries.

```
(module foo-eval
  (import ...)
  (eval (export-all)))
```

The standard distribution contains examples of such constructions. In particular, the multi-threading libraries `pthread` and `fthread` use this facility.

28.4 library and repl

It is possible to implement a "read-eval-print-loop" that is extended with the facilities implemented inside a library. In order to make the variables, functions, and classes of a library visible from the interpreter, the eval `library` module clause has to be used. (see Section 2.2 [Module Declaration], page 7) For instance, here is a module that implements a "repl" with the `format` facility available:

```
(module format-repl
   (eval (library bformat))
   (library bformat))

;; a dummy reference to a facility of the format library
(let ((dummy format))
   (repl))
```

Alternatively, libraries can be explicitly loaded using the `library-load` function such as:

```
(module format-repl)

;; a dummy reference to a facility of the format library
(let ((dummy format))
   (eval '(library-load 'bformat))
   (repl))
```

28.5 Building a library

Build Bigloo libraries require several steps that are explained in this section. This section shows how to create *static* and *dynamic* (or *shared*) libraries. However not that creating a dynamic library highly dependent on the host operating system. Users willing to create dynamic libraries on other operating systems should use the `api` directory of the Bigloo source code tree as an example.

- The first step is to build a *library heap*. This is achieved using a special compilation mode: `-mkaddheap -mkaddlib -addheap -heap-library <ident>`. That is, for your library you have to create a heap associated source file that imports all the binding

you want in your library. The heap source file must be *excluded* from the source files that will be used to build the host library.

Suppose we have a unique source file, `bformat.scm` for our library. The module clause of this source file is:

```
(module __bformat
   (export (bformat fmt::bstring . args)
           bformat:version))

(define (bformat fmt . args)
   (apply format (string-replace fmt #\% #\~) args))

(define bformat:version 1.0)
```

Prior to compiling the library, we have to create the heap associated file (let's name it `make_lib.scm`). This file could be:

```
(module __make_lib
   (import (__bformat "bformat.scm"))
   (eval (export-all)))
```

Building it is simple:

```
bigloo -unsafe -safee -q -mkaddheap -mkaddlib -heap-library bformat \
    make_lib.scm -addheap bformat.heap
```

The options `-mkaddheap` and `-mkaddlib` tell Bigloo that it is compiling an heap associated file. The option `-addheap` tells Bigloo the name of the heap file to be produced. The option `-heap-library` instructs the compiler for the library name to be included inside the heap file. This name is used for checking versions at run-time.

- The second step is to compile all the library source file. These compilation must be done using the `-mkaddlib` compilation mode. For example:

```
bigloo -O3 -unsafe -safee -mkaddlib       \
  -cc gcc -fsharing -q -rm                \
  -unsafev bformat.scm -o bformat_u.o -c
bigloo -O3 -mkaddlib -g -cg -cc gcc       \
  -fsharing -q -rm                        \
  -unsafev bformat.scm -o bformat.o -c
```

The first compilation produces the *unsafe* version the second the produced the *debugging* version.

- The third step is to build the host operating system libraries. There is no portable way to do this. This operation may looks like:

```
ar qcv libbigloobformat_s-1.0.a bformat.o
ranlib libbigloobformat_s-1.0.a
ld -G -o libbigloobformat_s-1.0.so bformat.o -lm -lc
ar qcv libbigloobformat_u-1.0.a bformat_u.o
ranlib libbigloobformat_u-1.0.a
ld -G -o libbigloobformat_u-1.0.so bformat_u.o -lm -lc
```

- The fourth step consist in creating the `bformat_es` and `bformat_eu` libraries for eval. For the unsafe version we use:

```
bigloo -O3 -unsafe -safee -mkaddlib       \
  -cc gcc -fsharing -q -rm                \
```

```
   -unsafev make_lib.scm -o make_lib.o -c
ld -G -o libbigloobformat_eu-1.0.so make_lib.o -lm -lc
ar qcv libbigloobformat_eu-1.0.a make_lib.o
ranlib libbigloobformat_eu-1.0.a
```

For the safe version we do:

```
bigloo -O3 -mkaddlib            \
   -cc gcc -fsharing -q -rm             \
   -unsafev make_lib.scm -o make_lib.o -c
ld -G -o libbigloobformat_es-1.0.so make_lib.o -lm -lc
ar qcv libbigloobformat_es-1.0.a make_lib.o
ranlib libbigloobformat_es-1.0.a
```

- The last step is to create an initialization file `bformat.init`:

```
(declare-library! 'bformat
   :version "1.0"
   :srfi '(bformat)
   :basename "bigloobformat"
   :module-init '__bformat
   :module-eval '__make_lib
   :class-init "bigloo.bformat.__bformat"
   :class-eval "bigloo.bformat.__make_lib")
```

At this time, you are ready to use your library. For that, let's assume the file `foo.scm`:

```
(module foo
   (library bformat))

(bigloo-library-path-set! (cons (pwd) (bigloo-library-path)))
(print (bformat "Library path: %a" (bigloo-library-path)))

(eval '(library-load 'bformat))
(repl)
```

It can be compiled and executed with:

```
bigloo foo.scm -L . -copt -L.
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH ./a.out
```

The Bigloo distribution contains library exemplars that should probably considered as a departure point for new libraries.

28.6 Library and modules

A Bigloo library may be composed of several Bigloo modules (even if in our example only one module was used). The modules composing the library are free to import each other. Nevertheless, someone designing a Bigloo library should be aware that Bigloo importation creates dependences between modules. A module `mod1` that imports a module `mod2` depends on `mod2` because `mod1` requires `mod2` to be initialized (i.e. `mod1` calls to the initialization function of `mod2`). The result is that using `import` clauses inside modules composing a library may create a lot of dependencies between the object files that are used to build the associated Unix library. Dependencies should be avoided because they make the Unix linkers unable to produce small stand-alone programs. Instead of `import` clauses, `use` clauses should be preferred. `Use` clauses do not create dependencies because a module `mod1` that `use`s a second module `mod2` does not require `mod2` to be initialized. Of course, it may

happen situations where the initialization is mandatory and thus, the `import` must not be replaced with a `use` clause. The source code of the Bigloo library makes use of `import` and `use` clauses. The Bigloo standard library should be studied as an example.

28.7 Library and macros

Bigloo libraries can export macros, expanders, and syntaxes but these must be handled carefully. Macros (these also applies to expanders and syntaxes) exported by modules are not visible by client code. Exported macros have to be placed inside the initialization file. For instance, if we change the definition of `bformat.init` file for:

```
(declare-library! 'bformat
   :version "1.0"
   :srfi '(bformat)
   :basename "bigloobformat"
   :module-init '__bformat
   :module-eval '__make_lib
   :class-init "bigloo.bformat.__bformat"
   :class-eval "bigloo.bformat.__make_lib")

(define-expander BFORMAT
   (lambda (x e)
      (match-case x
         ((?- (? (lambda (s) (and (string? s) (not (string-index s #\%))))) . ?a
)
          `(string-append ,@(cdr x)))
         (else
          `(bformat ,@(map (lambda (x) (e x e)) (cdr x)))))
```

At compile time the macro BFORMAT will be declared. Hence, we can change the definition of `foo.scm` for:

```
(module foo
   (library bformat))

(bigloo-library-path-set! (cons (pwd) (bigloo-library-path)))
(print (BFORMAT "library path: %a" (bigloo-library-path)))

(eval '(library-load 'bformat))
(repl)
```

28.8 A complete library example

For the means of an example let's suppose we want to design a Bigloo library for 2d points. That library is made of three implementation files: two C files, `cpoint.h` and `cpoint.c` and one Scheme file `spoint.scm`. Here are defined the three files:

`cpoint.h`:

```
struct point_2d {
   double x, y;
};
```

`cpoint.c`:

```
#include <stdio.h>
#include "cpoint.h"

int print_point_2d( struct point_2d *pt ) {
   printf( "<point-2d: %g, %g>", pt->x, pt->y );
```

```
}
```

`spoint.scm`:

```
(module __point
   (include "spoint.sch")
   (extern  (include "cpoint.h"))
   (export  (make-point::s-point_2d* ::double ::double)
            (print-point ::s-point_2d*)
            (point? ::obj)))

(define (make-point::s-point_2d* x::double y::double)
   (s-point_2d* x y))

(define (print-point p::s-point_2d*)
   (print_point_2d p))

(define (point? obj::obj)
   (s-point_2d*? obj)
   obj)
```

`makelib.scm`:

We want our library to be composed of the whole exported Scheme functions. Thus the file to build the heap library could look like:

```
(module __point_makelib
   (import __point)
   (eval (export-all)))
```

`point.init`: Let's suppose that the `point` library requires the `libposix` library. This means that any file linked with the `point` library needs to be also linked with the `posix` library. Furthermore, programs making use of the `point` library needs to include the `point.sch` file. That Scheme file needs in turn the C file `point.h` otherwise the produced C files won't compile. The need for the `libposix` library and for the `point.h` file may be specified inside the `point.init` file. For our current library, the `point.init` file could look like:

```
(declare-library! 'point
                  :basename "point"
                  :srfi '(point)
                  :eval-init '__point_makelib)

(set! *ld-options*
      (string-append "-L/usr/lib " *ld-options*))

(set! *bigloo-user-lib*
      (cons "-lm" *bigloo-user-lib*))

(set! *additional-include-foreign*
      (cons "cpoint.h" *additional-include-foreign*))

(define-macro (point x y)
   `(make-point ,x ,y))
```

This file updates some compilation variables (`*ld-options*`, `*bigloo-user-lib*`, `*additional-include-foreign*`) and defines a macro: `point`. Because the `point.init` file will be loaded each time a compilation require the `point` library is spawned, user code are allowed to use the `point` macro. Here is an example file making use of the `point` library:

`example.scm`

```
(module example)

(let ((p (point 2.9 3.5)))
   (print "point?: " (point? p))
   (print "point?: " (point? 4))
   (print-point p)
   (print "done..."))
```

To conclude that example here is the `Makefile` used to compile the `point` library, heap file and one example.

```
# bigloo flags
BIGLOO          = bigloo
RELEASE = `$(BIGLOO) -eval '(begin (print *bigloo-version*) (exit 0))'`
BHEAPFLAGS      = -unsafe -q -mkaddheap -mkaddlib -v2 -heap-library point
BCOMMONFLAGGS   = -mkaddlib -fsharing -q $(VERBOSE)        \
                  -copt '$(CCOMMONFLAGS)' -cc $(CC)
BSAFEFLAGS      = $(BCOMMONFLAGGS) -cg -O3 -g -cg -unsafev \
                  -eval '(set! *indent* 4)' -rm
BUNSAFEFLAGS    = $(BCOMMONFLAGS) -O4 -unsafe

# cigloo flags
CIGLOO          = cigloo

# cflags
CC              = gcc
CCOMMONFLAGS    = -I.
CSAFEFLAGS      = $(CCOMMONFLAGS)
CUNSAFEFLAGS    = $(CCOMMONFLAGS) -O2

# library objects
SAFE_OBJECT     = olib/spoint.o olib/cpoint.o
UNSAFE_OBJECT   = olib_u/spoint.o olib_u/cpoint.o

all: .afile heap lib example

.afile: spoint.scm makelib.scm
bglafile $^ > $@

heap: point.heap

point.heap: spoint.sch spoint.scm
$(BIGLOO) $(BHEAPFLAGS) makelib.scm -addheap point.heap

lib: lib_u lib.a

lib.a: olib $(SAFE_OBJECT)
ar qcv libpoint_s-$(RELEASE).a $(SAFE_OBJECT)
```

```
lib_u: olib_u $(UNSAFE_OBJECT)
ar qcv libpoint_u-$(RELEASE).a $(UNSAFE_OBJECT)

olib:
mkdir olib

olib_u:
mkdir olib_u

olib_u/spoint.o olib/spoint.o: spoint.scm
$(BIGLOO) $(BSAFEFLAGS) $(<F) -o $*.o -c

olib_u/cpoint.o olib/cpoint.o: cpoint.c
$(CC) $(CSAFEFLAGS) $(<F) -o $*.o -c

spoint.sch: cpoint.h cpoint.c
cigloo $^ > $@

example: heap lib
$(BIGLOO) -v2 -L . -library point \
          -static-bigloo example.scm -o example

clean:
-/bin/rm -f point.heap
-/bin/rm -f spoint.sch spoint.c
-/bin/rm -fr olib olib_u
-/bin/rm -f example example.c example.o
-/bin/rm -f libpoint_s-$(RELEASE).a libpoint_u-$(RELEASE).a
```

29 Extending the Runtime System

Custom Types types are not documented yet. This facility enables extension to the standard Bigloo runtime system. The current implementation of custom types is subject to change. It will be documented in coming releases.

30 SRFIs

Bigloo supports various SRFIs (Scheme Request For Implementation). Some of them are integrated in the Bigloo core libraries. Some others are implemented by the means of Bigloo libraries (see Chapter 28 [Bigloo Libraries], page 255). Only the first ones are described in the manual.

The current Bigloo core library support the following SRFIs:

- `srfi-0` (Conditional execution).
- `srfi-2` (AND-LET*: an AND with local bindings, a guarded LET* special form).
- `srfi-6` (Basic String Ports).
- `srfi-8` (Binding to multiple values).
- `srfi-9` (*Records* specification).
- `srfi-18` (Multithreading support).
- `srfi-22` (Script interpreter invocation).
- `srfi-28` (Basic Format Strings).
- `srfi-30` (Multi-line comments).
- `srfi-34` (Exception Handling for Programs).

30.1 SRFI 0

`cond-expand` [*clause*] [bigloo syntax]

The `cond-expand` form tests for the existence of features at macro-expansion time. It either expands into the body of one of its clauses or signals and error during syntactic processing. `cond-expand` expands into the body of the first clause whose feature requirement is currently satisfied (the `else` clause, if present, is selected if none of the previous clauses is selected).

A feature requirement has an obvious interpretation as a logical formula, where the variables have meaning *true* is the feature corresponding to the feature identifier, as specified in the *SRFI* registry, is in effect at the location of the `cond-expand` form, and *false* otherwise. A feature requirement is satisfied it its formula is true under this interpretation. The formula may make use of identifier, `and`, `or` and `not` operators.

Since Bigloo version 3.1b, `cond-expand` formula may use the new `library` operator that checks if a library exists and is available. Its syntax is: `(library <libname>)`.

Examples:

```
(write (cond-expand
          (srfi-0 (* 1 2))
          ((or (library fthread) (library pthread)) (- 4 1))
          (else (+ 3 4))))
   ⊣ 2

(cond-expand
   (bigloo (define (command-line-arguments) (command-line)))
   (else (define (command-line-arguments) '())))
```

The second example assumes that `bigloo` is an alias for the SRFI associated with the specification of Bigloo (i.e. the documentation for that Scheme system).

Since Bigloo 3.4b, `cond-expand` formula may use the new `config` operator that checks the value of a configuration entry. Its syntax is: `(config endianeness little-endian)`. This feature relies on the `bigloo-config` function. See Section 5.7 [System Programming], page 76, for additional details.

When writing portable code, the case used for the feature identifier should match the one in the SRFI registry. This is to ensure that the feature identifier will be correctly recognized whether or not the Scheme system is case-sensitive. To support case-insensitive Scheme systems, the feature identifiers in the SRFI registry are guaranteed to be unique even when ignoring the case.

In order to distinguish Bigloo versions, the following symbols are recognized in `cond-expand` forms.

- `bigloo`
- `bigloo<branch-release>`
- `bigloo<major-release>`
- `bigloo<major-release><minor-release>`

When finalizers have been configured, the two following symbols are recognized by `cond-expand`:

- `bigloo-finalizer`
- `bigloo-weakptr`

Bigloo implements differents SRFI for the compiler and the interpreter. Thus, their are two Bigloo SRFI registers. One for the compiler and one for the interpreter. Bigloo compiler SRFI register contains at least the following symbols:

- `srfi-0`
- `srfi-1`
- `srfi-2`
- `srfi-6`
- `srfi-8`
- `srfi-9`
- `srfi-22`
- `srfi-28`
- `srfi-30`

With respect to the currently used Bigloo back-end, one of these symbols is registered:

- `bigloo-c`
- `bigloo-jvm`

Bigloo compiler implements the following SRFI:

- `bigloo`
- `bigloo-compile`
- `bigloo<major-release>`
- `bigloo<major-release><minor-release>`

Then the `-g` flag is used, the Bigloo compiler additionally implements the SRFI:

- `bigloo-debug`

Bigloo interpreter implements the following SRFI:

- `bigloo`
- `bigloo-eval`
- `bigloo<major-release>`
- `bigloo<major-release><minor-release>`

When a library is used, the name of the library is added to the compiler SRFI register. That is:

```
(module foo
   (library srfi1))

(print (cond-expand (srfi1 'with-srfi1) (else 'nothing)))
   ⊣ 'with-srfi1
(print (eval '(cond-expand (srfi1 'with-srfi1) (else 'nothing))))
   ⊣ 'with-srfi1
```

A property representing actual integers bit size is defined:

- `bint<integers-bit-size>`
- `elong<exact-long-bit-size>`

The frequently defined values are:

- `bint30`: 32 bits architectures (e.g., x86)
- `elong32`: 32 bits architectures (e.g., x86)
- `bint32`: JVM
- `elong64`: JVM
- `bint61`: 64 bits architectures (e.g., x86_64)
- `elong64`: 64 bits architectures (e.g., x86_64)

Other values could be observed in the future. Note that the actual values of a particular setting can be obtained with:

```
(bigloo-config 'int-size)
(bigloo-config 'elong-size)
```

A configuration can be tested with:

- `config key value`

For instance:

```
(cond-expand
  ((and bigloo-c (config have-c99-stack-alloc #t)) ...)
  ...)
```

`register-srfi!` *srfi-name* [bigloo procedure]
`unregister-srfi!` *srfi-name* [bigloo procedure]
`unregister-eval-srfi!` *srfi-name* [bigloo procedure]
`register-compile-srfi!` *srfi-name* [bigloo procedure]
`unregister-compile-srfi!` *srfi-name* [bigloo procedure]

This argument *srfi-name* is a symbol. It registers *srfi-name* in the Bigloo interpreter SRFI register. This function must only be used when implementing a library. The

code of that library must contain one unique call to `register-eval-srfi!`. Let's suppose, for instance, a `format` library. The implementation for that library must contain an expression like:

The functions `unregister-XXX-srfi!` unregisters a srfi.

```
(register-eval-srfi! 'format)
```

Calling `(register-eval-srfi! name)` makes `name` supported by interpreted `cond-expand` forms.

Note: There is no `register-compiler-srfi!` because the compiler automatically registers SRFI when the `-library` flags are used. However, it exists several ways to tell the compiler that it actually supports some srfis when compiling some modules.

- The first way is to insert calls to `register-eval-srfi!` in the `.bigloorc` file (see Chapter 31 [Compiler Description], page 275).
- The second, is to use `option` (see Section 2.2 [Module Declaration], page 7) module clause, such as:

```
(module example
   ...
   (option (register-srfi! 'srfi-foobar)))

...
```

- The last way is to use the command line option `-srfi` (see Chapter 31 [Compiler Description], page 275).

30.2 SRFI 1

The SRFI 1 is implemented as a Bigloo library. Hence, in order to use the functions it provides, a module must import it.

```
(module ex
   (library srfi1))

(print (find-tail even? '(3 1 37 -8 -5 0 0)))
 ⇒ '(-8 -5 0 0))
```

30.3 SRFI 22

The SRFI 22 describes basic prerequisites for running Scheme programs as Unix scripts in a uniform way. A file (henceforth a *scipt*) conforming SRFI 22 has the following syntax:

```
<script>         ↦ <script prelude>? <program>
<script prelude> ↦ #! <space> <all but linebreak>* <linebreak>
```

A Scheme script interpreter loads the `<script>`. It ignores the script prelude and interprets the rest of the file according to the language dialect specified by the name of the interpreter.

The Scheme script interpreter may also load a different file after making a reasonable check that loading it is semantically equivalent to loading `<script>`. For example, the script interpreter may assume that a file with a related name (say, with an additional extension) is a compiled version of `<script>`.

30.3.1 An example of SRFI-22 script

Let us consider the following Bigloo script located in a file `foo.scm`:

```
#! /usr/bin/env ./execute
(module foo
   (main main))

(define (main argv)
   (print "foo: " argv))
```

Let us consider the following `execute` shell script:

```
$ cat > execute
#!/bin/sh
bigloo -i $*
```

Provided that `foo.scm` as the execute flag switched on, it is possible to *execute* it:

```
$ chmod u+x foo.scm
$ ./foo.scm
  ⊣ foo: (./foo.scm)
```

The same Bigloo module can be compiled and executed such as:

```
$ bigloo foo.scm
$ ./a.out
  ⊣ foo: (a.out)
```

30.3.2 Lazy compilation with SRFI-22

SRFI-22 can be used to implement *lazy* compilation. For instance, let us consider the following shell script:

```
$ cat > bgl
#!/bin/sh
SOURCEFILE=$1
case $SOURCEFILE in
   *.scm)
      OUTFILE=${SOURCEFILE%.scm}
      if ( bigloo -s -o $OUTFILE $SOURCEFILE ); then
         /bin/rm $OUTFILE.o
         shift
         ./$OUTFILE $@
      fi
      ;;
   *)
      echo Error: need a \*.scm file!
      ;;
esac
```

And the following Bigloo script:

```
#! /usr/bin/env ./bgl
(module foo
   (main main))

(define (main argv)
```

```
(print "foo: " argv))
```

When executed in the following way:

```
$ chmod u+x foo.scm
$ ./foo.scm
  ⊣ foo: (./foo.scm)
```

The Bigloo module `foo.scm` will first be compiled and then executed. Of course, one may consider more complex compiler drivers where it is first checked that the module is not already compiled.

31 Compiler description

31.1 C requirement

Instead of producing assembly code, Bigloo produces C code. This C code is ISO-C compliant [IsoC]. So, it is necessary to have an ISO-C compiler. The current version has been developed with `gcc` [Stallman95].

31.2 JVM requirement

In order to compile the Bigloo JVM back-end, you have to be provided with a JDK 1.2 or more recent (available at `http://www.javasoft.com`). The JVM must support for `-noverify` option because, by default, Bigloo produces JVM code that is not conform to the rules enforced by the Java byte code verifiers.

31.3 Linking

It is easier to use Bigloo for linking object files which have been compiled by Bigloo. An easy way to perform this operation is, after having compiled all the files using the `-c` option, to invoke Bigloo with the name of the compiled files.

When Bigloo is only given object file name as argument, it searches in the current directory and the directory named in the `*load-path*` list the Scheme source file in order to perform a correct link. Scheme source files are supposed to be ended by the suffix `.scm`. Additional suffixes can be added using the `-suffix` option. Hence, if source files are named foo1.sc and foo2.sc, a link command line could look like:

```
bigloo -suffix sc foo1.o foo2.o -o foo
```

Note: In order to understand how the Bigloo linkers operates and which libraries it uses, it might be useful to use the `-v2` option which unveil all the details of the compilation and the link.

31.4 The compiler environment and options

There are four ways to change the behaviour of Bigloo. Flags on the command line, the `option` module clause runtime-command file and environment variables See Chapter 2 [Modules], page 7. When the compiler is invoked, it first gets the environment variables, then it scans the runtime-command file and, at end, it parses the command line. If the same option is set many times, Bigloo uses the last one.

31.4.1 Efficiency

In order to get maximum speed, compile with the `-Obench` option. This will enable all compiler optimization options and disable dynamic type checks. To improve arithmetic performance see next section.

31.4.2 Stack allocation

When the `-fstack` flag is enabled, the compiler may automatically replace some heap allocations with stack allocations. This may improve performance because stack allocations are

handled more efficiently than heap allocations. On some cases, `-fstack` may also cause slow down or memory extra retentions. In this last case, when compile using `-fstack` the program will consume more memory. Unfortunately, this is nasty phenomenon is unpredictable (it depends on the nature of the source file).

31.4.3 Genericity of arithmetic procedures

By default, arithmetic procedures are generic. This means that it is allowed to use them with flonum and fixnum. This feature, of course, implies performances penalty. To improve performance, you may use specialized procedures (such as `+fx`, `=fx`, ... or `+fl`, `=fl`, ...) but, it is possible to suppress the genericity and to make all generic arithmetic procedures (`=` for example) fixnum ones. For this you must use the compiler option `-farithmetic`, or add the following module clause `(option (set! *genericity* #f))` in your module declaration.

31.4.4 Safety

It is possible to generate *safe* or *unsafe* code. The safety's scope is `type`, `arity`, `version` and `range`. Let's see an example:

```
(define (foo f v indice)
   (car (f (vector-ref v indice))))
```

In safe mode, the result of the compilation will be:

```
(define (foo f v indice)
  (let ((pair
        (if (and (procedure? f)
                 ;; type check
                 (= (procedure-arity f) 1))
                 ;; arity check
            (if (vector? v)
                ;; type check
                (if (and (integer? k)
                         ;; type check
                         (>= k 0)
                         ;; range check
                         (< k (vector-length v)))
                         ;; range check
                    (f (vector-ref v indice))
                    (error ...))
                (error ...))
            (error ...))))
    (if (pair? pair)
        ;; type check
        (car pair)
        (error ...))))
```

It is possible to remove some or all safe checks. For example, here is the result of the compilation where safe check on types have been removed:

```
(define (foo f v indice)
  (let ((pair (if (= (procedure-arity f) 1)
                  ;; arity check
                  (if (and (>= k 0)
                           ;; range check
                           (< k (vector-length v)))
                           ;; range check
                      (f (vector-ref v indice))
```

```
            (error ...))
          (error ...))))
   (car pair)))
```

31.4.5 The runtime-command file

Each Bigloo's user can use a special configuration file. This file must be named "`.bigloorc`" or "`~/.bigloorc`". Bigloo tries to load one of these in this order. This file is a Scheme file. Bigloo exports variables which allow the user to change the behavior of the compiler. All these variables can be checked using the -help2 option.

The Bigloo's runtime command file is read before the arguments are parsed.

31.4.6 The Bigloo command line

If no input file is specified, Bigloo enters its interpreter. Here is the exhaustive list of Bigloo options and configuration variables:

usage: bigloo [options] [name.suf]

Misc:

-	Read source code on current input channel
-help,–help	This help message
-help2	The exhaustive help message
-help-manual	The help message formatted for the manual
-o FILE	Name the output FILE
–to-stdout	Write C code on current output channel
-c	Suppress linking and produce a .o file
-y	Generate a shared library
-suffix SUFFIX	Recognize suffix as Scheme source
-afile FILE	Name of the access file
-access MODULE FILE	Set access between module and file
-jfile FILE	Name of the Jvm package file
-jadd MODULE QTYPE	Set JVM qualifed type name for module
-main FUN	Set the main function
-with MODULE	Import addition module
-multiple-inclusion	Enables multiple inclusions of the Bigloo includes
-library LIBRARY	Compile/link with additional Bigloo library
-srfi SRFI	Declares srfi support
-dload-sym	Emit a Bigloo dynamic loading entry point
-dload-init-sym NAME	Emit a Bigloo dynamic loading entry point, named NAME
-dload-init-gc	For GC initialization for dynamic code
-heapsize SIZE	Set the initial heap size value (in megabyte)

Configuration and path:

-version	The current release
-revision	The current release (short format)
-query	Dump the current configuration

```
  -q                          Do not load any rc file
  -eval STRING                Evaluate STRING before compiling
  -load FILE                  Load FILE before compiling
  -I DIR                      Add DIR to the load path
  -lib-dir DIR                Set lib-path to DIR
  -L NAME                     Set additional library path
  -lib-version VERSION        Set the Bigloo library version
  -libgc-version VERSION      Set the Bigloo GC library version
  -libgc GC                   Use the given GC library

Back-end:
  -native                     Compile module to native object file (via C)
  -jvm                        Compile module to JVM .class files
  -saw                        Cut the AST in the saw-mill
  -no-saw                     Disable saw back-ends
  -i                          Interprete module

Dialect:
  -snow                       Compiles a snow source code
  -scmpkg,-spi                Compiles a ScmPkg source code
  -nil                        Evaluate '() as #f in 'if' expression
  -call/cc                    Enable call/cc function
  -hygien                     Obsolete (R5rs macros are always supported)
  -fidentifier-syntax SYNTAX  Identifiers syntax "r5rs" (default) or "bigloo"
  -fno-reflection             Deprecated
  +fno-reflection             Deprecated
  -fclass-nil                 Deprecated
  -fno-class-nil              Deprecated
  -farithmetic                Suppress genericity of arithmetic operators
  -farithmetic-overflow       Suppress arithmetic overflow checks
  -fno-arithmetic-overflow    Enable arithmetic overflow checks
  -fcase-sensitive            Case sensitive reader (default)
  -fcase-insensitive          Case insensitive reader (downcase symbols)
  -fallow-type-redefinition   allow type redifinition
  -runtime-code-patching      Enable runtime code patching
  -no-runtime-code-patching   Disable runtime code patching

Optimization:
  -Obench                     Benchmarking mode
  -O[0..6]                    Optimization modes
  -ftagged-fxop               Enable tagged fix-ops optimization
  -fno-tagged-fxop            Disable tagged fix-ops optimization
  -fcfa                       Enable CFA
  -fno-cfa                    Disable CFA
  -fcfa-arithmetic            Enable arithmetic spec. (see -farithmetic-overflow)
  -fno-cfa-arithmetic         Disable arithmetic spec.
  -fcfa-arithmetic-fixnum     Enable fixnum arithmetic spec.
```

```
  -fno-cfa-arithmetic-fixnum        Disable fixnum arithmetic spec.
  -fcfa-arithmetic-flonum          Enable flonum arithmetic spec. (enabled from -O2)
  -fno-cfa-arithmetic-flonum        Disable flonum arithmetic spec.
  -fcfa-tracking                  Enable CFA tracking (enabled from -O2)
  -fno-cfa-tracking                Disable CFA tracking
  -fcfa-pair                     Enable CFA pairs approximations
  -fno-cfa-pair                   Disable CFA pairs approximations
 -fcfa-unbox-closure-args         Enable CFA unboxed closure args (enabled from -O2)
  -fno-cfa-unbox-closure-args       Disable CFA unboxed closure args
  -fno-cfa-local-function          Disable CFA local function tracking
  -funroll-loop                   Enable loop unrolling (enabled from -O3)
  -fno-unroll-loop                 Disable loop unrolling
  -fno-loop-inlining               Disable loop inlining
  -floop-inlining                 Enable loop inlining (default)
  -fno-inlining                   Disable inline optimization
  -fno-user-inlining               Disable user inline optimization
  -fisa                         Inline isa? type predicate
  -fno-isa                       Inline isa? type predicate
  -fbeta-reduce                    Enable simple beta reduction (enabled from -O2)
  -fno-beta-reduce                 Disable simple beta reduction
  -fdataflow                      Enable dataflow optimizations (enabled from -O)
  -fno-dataflow                    Disable dataflow optimizations
  -fdataflow-for-errors           Enable dataflow optimizations for improviing type er-
ror messages
  -fno-dataflow-for-errors        Disable dataflow optimizations for improviing type er-
ror messages
  -fdataflow-types                Enable type dataflow optimizations (enabled from -O2)
  -fno-dataflow-types              Disable type dataflow optimizations
  -finitflow                     Enable init flow
  -fno-initflow                   Disable init flow
  -fsync-failsafe                 Enable failsafe synchronize optimization
  -fno-sync-failsafe               Disable failsafe synchronize optimization
  -fO-macro                        Enable Optimization macro (default)
  -fno-O-macro                     Disable Optimization macro
  -fglobal-tailc                  Enable global tail-call optimization
  -fno-global-tailc                Disable global tail-call optimization
  -freturn                       Enable set-exit replacement with return
  -fno-return                     Disable set-exit replacement
  -freturn-goto                    Enable local set-exit replacement with return
  -fno-return-goto                 Disable local set-exit replacement
  -fsaw-realloc                   Enable saw register re-allocation
  -fsaw-regalloc                  Enable saw register allocation
  -fno-saw-regalloc                Disable saw register allocation
  -fsaw-bbv                       Enable saw basic-blocks versionning
  -fno-saw-bbv                     Disable saw basic-blocks versionning
  -fsaw-regalloc-msize SIZE          Set the register allocation body size limit
  -fsaw-regalloc-fun NAME            Allocate registers on this very function
```

```
  -fno-saw-regalloc-fun NAME      Don't allocate registers on this very function
  -fsaw-regalloc-onexpr           Allocate registers on expressions
  -fno-saw-regalloc-onexpr        Don't allocate registers on expressions
  -fsaw-spill                     Enable saw spill optimization

Safety:
  -unsafe[atrsvleh]         Don't check [type/arity/range/struct/version/library/eval/heap]
  -safe[atrsvle]            Enforce check [type/arity/range/struct/version/library/eval]

Debug:
  -glines                   Emit # line directives
  -gbdb-no-line             Don't emit # line directives
  -gbdb[23]                 Compile with bdb debug informations
  -gself                    Enables self compiler debug options
  -gmodule                  Debug module initialization
  -gerror-localization      Localize error calls in the source code
  -gno-error-localization   Don't localize error calls in the source code
  -gjvm                     Annote JVM classes for debug
  -gtrace[12]               Producing stack traces
  -g[234]                   Produce Bigloo debug informations
  -cg                       Compile C files with debug option
  -export-all               Eval export-all all routines
  -export-exports           Eval export-exports all routines
  -export-mutable           Enables Eval redefinition of all "::obj" routines

Profiling:
  -p[2g]                    Compile for cpu profiling
  -pmem[level]              Compile for memory profiling
  -psync                    Profile synchronize expr (see $exitd-mutex-profile)

Verbosity:
  -s                        Be silent and inhibit all warning messages
  -v[23]                    Be verbose
  -hello                    Say hello
  -no-hello                 Dont' say hello even in verbose mode
  -w                        Inhibit all warning messages
  -wslots                   Inhibit overriden slots warning messages
  -Wvariables               Enable overriden variable warning messages
  -Wtypes                   Enable type check warning messages
  -Wslot                    Enable default slot value warning messages
  -Wno-slot                 Disable default slot value warning messages
  -Wall                     Warn about all possible type errors
  -Werror=type              Treat type warnings as error

Compilation modes:
  <-/+>rm                   Don't or force removing .c or .il files
  -extend NAME              Extend the compiler
```

```
  -fsharing                 Attempt to share constant data
  -fno-sharing              Do not attempt to share constant data
  -fmco                     Produce an .mco file
  -fmco-include-path DIR        Add dir to mco C include path

Native specific options:
  -cc COMPILER                Specify the C compiler
  -stdc                     Generate strict ISO C code
  -copt STRING                Invoke cc with STRING
  -cheader STRING             C header
  -cfoot STRING               C foot
  -rpath PATH                 Add C runtime-path (rpath)
  -ldopt STRING               Invoke ld with STRING
  -ldpostopt STRING           Invoke ld with STRING (end of arguments)
  –force-cc-o               Force the C compiler to use -o instead of mv
  -ld-relative              Link using -l notation for libraries (default)
  -ld-absolute              Link using absolute path names for libraries
  -static-bigloo            Link with the static bigloo library
  -static-all-bigloo        Link with static version of all bigloo libraries
  -ld-libs1                 Add once user libraries when linking
  -ld-libs2                 Add twice user libraries when linking (default)
  -lLIBRARY                   Link with host library
  -auto-link-main           Enable main generation when needed for linking
  -no-auto-link-main          Disable main generation
  –force-gc-roots           Register global variables as GC roots

Jvm specific options:
  -jvm-shell SHELL            Shell for JVM scripts ("sh", "msdos")
 -jvm-purify                Produce byte code verifier compliant JVM code (default)
  -no-jvm-purify            Don't care about JVM code verifier
  -jvm-mainclass CLASS        JVM main class
  -jvm-classpath PATH         JVM application classpath
  -jvm-bigloo-classpath P     JVM Bigloo rts classpath
  -jvm-path-separator SEP     Set the JVM classpath separator
  -jvm-directory NAME         Directory where to store class files.
  -jvm-catch-errors         Catch internal JVM errors
  -no-jvm-catch-errors      Don't catch internal JVM errors
  -jvm-jarpath NAME           Set the JVM classpath for the produced jar file
   -jvm-cinit-module              Enable JVM class constructors to initili-
aze bigloo modules
   -no-jvm-cinit-module           Disable JVM class constructors to initili-
aze bigloo modules
   -jvm-char-info               Generate char info for the debugger (in addi-
tion to line info)
  -no-jvm-char-info         Do not generate char info for the debugger
  -fjvm-inlining            Enable JVM back-end inlining
  -fjvm-constr-inlining       Enable JVM back-end inlining for constructors
```

```
  -fno-jvm-inlining          Disable JVM back-end inlining
  -fno-jvm-constr-inlining   Disable JVM back-end inlining for constructors
  -fjvm-peephole             Enable JVM back-end peephole
  -fno-jvm-peephole          Disable JVM back-end peephole
  -fjvm-branch               Enable JVM back-end branch
  -fno-jvm-branch            Disable JVM back-end branch
  -fjvm-fasteq               EQ? no longer works on integers (use =FX)
  -fno-jvm-fasteq            Disable JVM back-end fasteq transformation
  -jvm-env VAR               Make the shell variable visible to GETENV
  -jvm-jar                   Enable JVM jar files generation
  -no-jvm-jar                Disable JVM jar files generation (default)
  -jvm-java FILE             Use FILE as JVM
  -jvm-opt STRING            JVM invocation option

Traces:
  -t[2|3|4]                  Generate a trace file (*)
  +tPASS                     Force pass to be traced
  -shape[mktTalun]           Some debugging tools (private)

Compilation stages:
  -mco                       Stop after .mco production
  -syntax                    Stop after the syntax stage (see -hygiene)
  -expand                    Stop after the preprocessing stage
  -expand-module             Produce the expanded module clause
  -ast                       Stop after the ast construction stage
  -syntax-check              Stop after checking syntax
  -bdb-spread-obj            Stop after the bdb obj spread stage
  -trace                     Stop after the trace pass
  -callcc                    Stop after the callcc pass
  -bivalue                   Stop after the bivaluation stage
  -inline                    Stop after the inlining stage
  -inline+                   Stop after the 2nd inlining stage
  -beta                      Stop after the constant beta reduction stage
  -fail                      Stop after the failure replacement stage
  -abound                    Stop after the array bound checking stage
  -initflow                  Stop after the type initflow stage
  -narrow                    Stop after the scope narrowing stage
  -tlift                     Stop after the type lifting stage
  -dataflow                  Stop after the type dataflow stage
  -dataflow+                 Stop after the second type dataflow stage
  -dataflow++                Stop after the third type dataflow stage
  -fuse                      Stop after the fuse stage
  -user                      Stop after the user pass
  -fxop                      Stop after the fx-ops optimization
  -coerce                    Stop after the type coercing stage
  -effect                    Stop after the effect stage
  -effect+                   Stop after the 2nd effect stage
```

```
  -reduce                          Stop after the reduction opt. stage
  -reduce+                         Stop after the 2nd reduction opt. stage
  -reduce-                         Stop after the very first reduction stage
  -assert                          Stop after the assertions stage
  -cfa                             Stop after the cfa stage
  -closure                         Stop after the globalization stage
  -recovery                        Stop after the type recovery stage
  -bdb                             Stop after the Bdb code production
  -cnst                            Stop after the constant allocation
  -integrate                       Stop after the integration stage
  -tailc                           Stop after the tailc stage
  -return                          Stop after the return stage
  -isa                             Stop after the isa stage
  -init                            Stop after the initialization construction stage
  -classgen                        Produce an include file for class accessors
  -egen                            Produce an include file for effects (requires -saw)
  -hgen                            Produce a C header file with class definitions
  -cgen                            Do not C compile and produce a .c file
  -indent                          Produce an indented .c file
  -jvmas                           Produce a JVM .jas file

Constant initialization:
  -init-[lib|read|intern]          Constants initialization mode
  -init-object-[legacy|staged]     Object system initialization

Bootstrap and setup:
  -mklib                           Compile a library module
  -mkaddlib                        Compile an additional library module
  -mkheap                          Build an heap file
  -mkaddheap                       Build an additional heap file
  -mkdistrib                       Compile a main file for a distribution
  –license                         Display the Bigloo license and exit
  -LICENSE                         Add the license to the generated C files
  -heap NAME                       Specify an heap file (or #f to not load heap)
  -heap-library LIB                The library the heap belongs to
  -dump-heap NAME                  Dump the content of a heap
  -addheap NAME                    Specify an additional heap file
  -fread-internal                  Read source from binary interned file
  -fread-internal-src              Read source only from binary interned file
  -fread-internal-src-file-name NAME Set fake source file name
  -fread-plain                     Read source from plain text file
  -target LANG                     DON'T USE, (see -native, -jvm)

Shell Variables:
  - TMPDIR
    temporary directory (default "/tmp")
  - BIGLOOLIB
```

libraries' directory
- BIGLOOHEAP
  the initial heap size in megabytes (4 MB by default)
- BIGLOOSTACKDEPTH
  the error stack depth printing
- BIGLOOLIVEPROCESS
  the maximum number of Bigloo live processes
- BIGLOOTRACE
  list of active traces

Runtime Command file:
- ~/.bigloorc

---

* : only available in developing mode
. : option enabled from -O3 mode

Bigloo Control Variables:
All the Bigloo control variables can be changed from the interpreter, by the means of the '-eval' option, or using the module clause 'option'. For instance the option "-eval '(set! *strip* #t)'" will set the variable '*strip*' to the value '#t'.
These variables are:

- *access-file-default* :
  The default access file name
  default: ".afile"
- *access-files* :
  The access file names
  default: ()
- *additional-bigloo-libraries* :
  The user extra Bigloo libraries
  default: ()
- *additional-bigloo-zips* :
  The user extra Bigloo Zip files
  default: ()
- *additional-heap-name* :
  A name of an additional heap file name to be build
  default: #f
- *additional-heap-names* :
  A list of Bigloo additional heap file name
  default: ()
- *additional-include-foreign* :
  The additional C included files

default: ()
- *allow-type-redefinition* :
  If true, allow type redefinitions
  default: #f
- *ast-case-sensitive* :
  Case sensitivity
  default: #t
- *auto-link-main* :
  Enable automatically a main generation when linking
  default: #t
- *auto-mode* :
  auto-mode (extend mode) list
default: (("ml" . "caml") ("mli" . "caml") ("oon" . "meroon") ("snow" . "snow") ("spi" . "pkgcomp"))
- *bdb-debug* :
  Bdb debugging mode
  default: 0
- *bigloo-abort?* :
  Do we have the bigloo-abort function in executables?
  default: #f
- *bigloo-lib* :
  The Bigloo library
  default: bigloo
- *bigloo-libraries-c-setup* :
  A list of C functions to be called when starting the application
  default: ()
- *bigloo-licensing?* :
  Add the Bigloo license ?
  default: #f
- *bigloo-name* :
  The Bigloo name
  default: "Bigloo (4.3g)"
- *bigloo-specific-version* :
  The Bigloo specific version
  default: ""
- *bigloo-tmp* :
  The tmp directory name
  default: "/tmp"
- *bigloo-user-lib* :
  The user extra C libraries
  default: ("-ldl" "-lresolv" "-lgmp" "-lunistring" "-lpcre" "-lm")
- *bigloo-version* :
  The Bigloo major release number
  default: "4.3g"
- *bmem-profiling* :
  Instrument code for bmem profiling
  default: #f
- *c-debug* :

C debugging mode?
default: #f
- *c-debug-lines-info* :
Emit # line directives
default: #f
- *c-debug-option* :
cc debugging option
default: "-g"
- *c-files* :
The C source files
default: ()
- *c-object-file-extension* :
The C object file extension
default: "o"
- *c-split-string* :
C split long strings
default: #f
- *c-suffix* :
C legal suffixes
default: ("c")
- *c-user-foot* :
C foot
default: ()
- *c-user-header* :
C header
default: ()
- *call/cc?* :
Shall we enable call/cc?
default: #f
- *cc* :
The C compiler
default: "gcc"
- *cc-move* :
Use mv instead of -o when C compiling
default: #t
- *cc-o-option* :
The C compiler -o option
default: "-o "
- *cc-options* :
cc options
default: ("")
- *cc-style* :
The C compiler style
default: "gcc"
- *cflags* :
The C compiler option
default: " -Wpointer-arith -Wswitch -Wtrigraphs -DBGL_BOOTCONFIG "

- *cflags-optim* :
  The C compiler optimization option
  default: "-O3"
- *cflags-prof* :
  The C compiler profiling option
  default: "-pg -fno-inline -Wpointer-arith -Wswitch -Wtrigraphs -DBGL_BOOTCONFIG "
- *cflags-rpath* :
  The C compiler rpath option
  default: ("/home/serrano/prgm/project/bigloo/bigloo/lib/bigloo/4.3g")
- *compiler-debug* :
  Debugging level
  default: 0
- *compiler-debug-trace* :
  Debugging trace level
  default: 0
- *compiler-sharing-debug?* :
  Compiler self sharing debug
  default: #f
- *compiler-stack-debug?* :
  Compiler self stack trace debug
  default: #f
- *compiler-type-debug?* :
  Compiler self type debug
  default: #f
- *csharp-suffix* :
  C# legal suffixes
  default: ("cs")
- *debug-module* :
  Module initilazation debugging
  default: 0
- *default-lib-dir* :
  Depreacted, don't use
  default: "/home/serrano/prgm/project/bigloo/bigloo/lib/bigloo/4.3g"
- *dest* :
  The target name
  default: #f
- *dlopen-init* :
  Emit a standard Bigloo dynamic loading init entry point
  default: #f
- *dlopen-init-gc* :
  Emit a standard GC init call when initialization the module
  default: #f
- *double-ld-libs?* :
  Do we include the additional user libraries twice?
  default: #t
- *error-localization* :
  Localize error calls in the source code

default: #f
- *eval-options* :
A user variable to store dynamic command line options
default: ()
- *extend-entry* :
Extend entry
default: #f
- *garbage-collector* :
The garbage collector
default: boehm
- *gc-custom?* :
Are we using a custom GC library?
default: #t
- *gc-force-register-roots?* :
Force GC roots registration for global variables (for experts only)
default: #t
- *gc-lib* :
The Gc library
default: bigloogc
- *global-tail-call?* :
Do we apply the self-global-tail-call stage?
default: #f
- *heap-base-name* :
The Bigloo heap base name
default: "bigloo"
- *heap-dump-names* :
The name of the heap to be dumped
default: ()
- *heap-jvm-name* :
The Bigloo heap file name for the JVM backend
default: "bigloo.jheap"
- *heap-library* :
The library the heap belongs to
default: bigloo
- *heap-name* :
The Bigloo heap file name
default: "bigloo.heap"
- *hello* :
Say hello (when verbose)
default: #f
- *include-foreign* :
The C included files
default: ("bigloo.h")
- *include-multiple* :
Enable/disable multiple inclusion of same file
default: #f
- *indent* :

The name of the C beautifier
default: "indent -npro -bap -bad -nbc -bl -ncdb -nce -nfc1 -ip0 -nlp -npcs -nsc -nsob -cli0.5 -di0 -l80 -d1 -c0 -ts2 -st"
- *init-mode* :
Module initialization mode
default: read
- *inlining-kfactor* :
Inlining growth factor
default: #<procedure:55b60df83c60.1>
- *inlining-reduce-kfactor* :
Inlinine growth factor reductor
default: #<procedure:55b60df83ba0.1>
- *inlining?* :
Inlining optimization
default: #t
- *interpreter* :
Shall we interprete the source file?
default: #f
- *jvm-bigloo-classpath* :
JVM Bigloo classpath
default: #f
- *jvm-catch* :
Catch internal errors
default: #t
- *jvm-cinit-module* :
Enable JVM class constructors to initiliaze bigloo modules
default: #f
- *jvm-classpath* :
JVM classpath
default: "."
- *jvm-debug* :
JVM debugging mode?
default: #f
- *jvm-directory* :
JVM object directory
default: #f
- *jvm-env* :
List of environment variables to be available in the compiled code
default: ()
- *jvm-foreign-class-id* :
The identifier of the Jlib foreign class
default: foreign
- *jvm-foreign-class-name* :
The name of the Jlib foreign class
default: "bigloo.foreign"
- *jvm-jar?* :
Enable/disable a JAR file production for the JVM back-end

default: #f
- *jvm-jarpath* :
JVM jarpath
default: #f
- *jvm-java* :
JVM to be used to run Java programs
default: "java"
- *jvm-mainclass* :
JVM main class
default: #f
- *jvm-options* :
JVM options
default: ""
- *jvm-path-separator* :
JVM classpath
default: #f
- *jvm-shell* :
Shell to be used when producing JVM run scripts
default: "sh"
- *ld-debug-option* :
The C linker debugging option
default: "-g "
- *ld-library-dir* :
Depreacted, don't use
default: "/home/serrano/prgm/project/bigloo/bigloo/lib/bigloo/4.3g"
- *ld-o-option* :
The C linker -o option
default: "-o "
- *ld-optim-flags* :
The C linker optimization flags
default: ""
- *ld-options* :
ld options
default: " "
- *ld-post-options* :
ld post options
default: ("")
- *ld-relative* :
Relative or absolute path names for libraries
default: #t
- *ld-style* :
ld style
default: "gcc"
- *lib-dir* :
The lib dir path
default: ("." "/home/serrano/prgm/project/bigloo/bigloo/lib/bigloo/4.3g")
- *lib-mode* :

Lib-mode compilation?
default: #f
- *lib-src-dir* :
The lib dir path
default: "runtime"
- *load-path* :
The load path
default: ("." "/home/serrano/prgm/project/bigloo/bigloo/lib/bigloo/4.3g")
- *max-c-foreign-arity* :
Max C function arity
default: 16
- *max-c-token-length* :
Max C token length
default: 1024
- *mco-include-path* :
Module checksum C include path
default: (".")
- *mco-suffix* :
Module checksum object legal suffixes
default: ("mco")
- *module-checksum-object?* :
Produce a module checksum object (.mco)
default: #f
- *multi-threaded-gc?* :
Are we using a multi-threaded GC?
default: #f
- *o-files* :
The additional obect files
default: ()
- *obj-suffix* :
Object legal suffixes
default: ("o" "a" "so")
- *object-init-mode* :
Object initialization mode
default: stagged
- *optim* :
Optimization level
default: 0
- *optim-O-macro?* :
Enable optimization by macro-expansion
default: #f
- *optim-atom-inlining?* :
Skip atom in inlining parameter counting
default: #t
- *optim-cfa-apply-tracking?* :
Track values across apply
default: #f

- *optim-cfa-fixnum-arithmetic?* :
  Enable refined fixnum arithmetic specialization
  default: #f
- *optim-cfa-flonum-arithmetic?* :
  Enable refined flonum arithmetic specialization
  default: #f
- *optim-cfa-force-loose-local-function?* :
  Force loosing local function approximations (for debugging)
  default: #f
- *optim-cfa-free-var-tracking?* :
  Enable closure free-variables specialization
  default: #f
- *optim-cfa-pair-quote-max-length* :
  Maximum length for pair literal tracking
  default: 4
- *optim-cfa-pair?* :
  Track values across pairs
  default: #f
- *optim-cfa-unbox-closure-args* :
  Unbox closure arguments
  default: #f
- *optim-cfa?* :
  Enable CFA
  default: #t
- *optim-dataflow-for-errors?* :
  Enable simple dataflow optimization for eliminating bad error messages
  default: #t
- *optim-dataflow-types?* :
  Enable dataflow optimization for types
  default: #f
- *optim-dataflow?* :
  Enable simple dataflow optimization
  default: #f
- *optim-initflow?* :
  Enable initflow optimization for global variables
  default: #f
- *optim-integrate?* :
  Enable function integration (closure analysis)
  default: #t
- *optim-isa?* :
  Enable isa type tests optimization (inlining)
  default: #f
- *optim-jvm* :
  Enable optimization by inlining jvm code
  default: 0
- *optim-jvm-branch* :
  Enable JVM branch tensioning

default: 0
- *optim-jvm-constructor-inlining* :
Enable JVM inlining for constructors
default: 0
- *optim-jvm-fasteq* :
EQ? no longer works on integers (use =FX instead)
default: #f
- *optim-jvm-inlining* :
Enable JVM inlining
default: 0
- *optim-jvm-peephole* :
Enable JVM peephole optimization
default: 0
- *optim-loop-inlining?* :
Loop inlining optimization
default: #t
- *optim-patch?* :
Optimize self modifying code
default: #f
- *optim-reduce-beta?* :
Enable simple beta reduction
default: #f
- *optim-return-goto?* :
Optimize set-exit by enabling local return
default: #f
- *optim-return?* :
Optimize set-exit used as return
default: #f
- *optim-symbol-case* :
Optimize case forms descrimining on symbols only
default: #f
- *optim-sync-failsafe?* :
Enable failsafe synchronize optimization
default: #f
- *optim-tagged-fxop?* :
Optimize tagged fixnum operations
default: #f
- *optim-unroll-loop?* :
Loop unrolling optimization
default: #unspecified
- *pass* :
Stop after the pass
default: ld
- *patch-support* :
Enables/disables patch (runtime code modification) support
default: #f
- *pre-processor* :

An optional function that pre-processes the source file
default: #<procedure:55b60df824f0.1>
- *prof-table-name* :
Bprof translation table file name
default: "bmon.out"
- *profile-library* :
Use the profiled library version
default: #f
- *profile-mode* :
Bigloo profile mode
default: 0
- *purify* :
Produce byte code verifier compliant JVM code
default: #t
- *qualified-type-file* :
The qualifed-type association file name
default: #f
- *qualified-type-file-default* :
The qualifed-type association file name
default: ".jfile"
- *reader* :
The way the reader reads input file ('plain or 'intern)
default: plain
- *rm-tmp-files* :
Shall the .c and .il produced files be removed?
default: #t
- *saw* :
Do we go to the saw-mill?
default: #f
- *saw-bbv?* :
Enable/disable saw basic-blocks versionning
default: #f
- *saw-no-register-allocation-functions* :
The list of functions disabling register allocation
default: ()
- *saw-register-allocation-functions* :
The list of functions allowing register allocation
default: ()
- *saw-register-allocation-max-size* :
Max function size for optimizing the register allocation
default: 4000
- *saw-register-allocation-onexpression?* :
Enable/disable saw register allocation on expression
default: #f
- *saw-register-allocation?* :
Enable/disable saw register allocation
default: #f

- *saw-register-reallocation?* :
  Enable/disable saw register re-allocation
  default: #f
- *saw-spill* :
  Enable saw spill optimization
  default: #f
- *shared-cnst?* :
  Shared constant compilation?
  default: #t
- *shell* :
  The shell to exec C compilations
  default: "/bin/sh"
- *src-files* :
  The sources files
  default: ()
- *src-suffix* :
  Scheme legal suffixes
  default: ("scm" "bgl")
- *startup-file* :
  A startup file for the interpreter
  default: #f
- *static-all-bigloo?* :
  Do we use the static version of all Bigloo libraries?
  default: #f
- *static-bigloo?* :
  Do we use the static Bigloo library
  default: #f
- *stdc* :
  Shall we produce ISO C?
  default: #f
- *strip* :
  Shall we strip the executable?
  default: #t
- *sync-profiling* :
  Instrument code for synchronize profiling
  default: #f
- *target-language* :
  The target language (either c, c-saw, jvm, or .net)
  default: native
- *trace-name* :
  Trace file name
  default: "trace"
- *trace-write-length* :
  Trace dumping max level
  default: 80
- *unsafe-arity* :
  Runtime type arity safety

default: #f
- *unsafe-eval* :
Disable type checking for eval functions
default: #f
- *unsafe-heap* :
Disable heap version checking
default: #f
- *unsafe-library* :
Use the unsafe library version
default: #f
- *unsafe-range* :
Runtime range safety
default: #f
- *unsafe-struct* :
Runtime struct range safety
default: #f
- *unsafe-type* :
Runtime type safety
default: #f
- *unsafe-version* :
Module version safety
default: #f
- *user-heap-size* :
Heap size (in MegaByte) or #f for default value
default: 0
- *user-inlining?* :
User inlining optimization
default: #t
- *user-pass* :
The user specific compilation pass
default: #unspecified
- *verbose* :
The verbosity level
default: 0
- *warning-default-slot-value* :
Set to #t to warn about non-inlinable default slot values
default: #f
- *warning-overriden-slots* :
Set to #t to warn about virtual slot overriding
default: #t
- *warning-overriden-variables* :
Set to #t to warn about variable overriding
default: #f
- *warning-type-error* :
Set to #t to treat type warnigns as error
default: #f
- *warning-types* :

  Set to #t to warn about type checks
  default: #f
- *with-files* :
  The additional modules
  default: ()

32 Cross Compilation

Bigloo is very portable and can be cross-compiled for most Posix-platforms. As long as there exists a C (cross-)compiler for the platform and the garbage collector is supported on the targeted platform there is a good chance that Bigloo or Bigloo-compiled programs will run on the platform.

This chapter describes how to cross-compile Bigloo with a C cross-compiler. Following established conventions we will call the platform where the compiled programs should run the *Host* platform and we will call the build platform where we actually compile the programs the *Build* platform.

32.1 Introduction

We assume that the host- and build-system are not the same, and that there exists a C cross-compiler `CC` running on the build system producing executables for the host system.

In order to execute programs on the host, it is however not sufficient to simply compile Bigloo-produced programs with this compiler. Indeed, these programs depend on the Bigloo-library which thus has to exist on the host-platform.

Building a cross-compilation environment is done in two phases:

- Build a Bigloo for the build-platform. Usually this is a given.
- Build the Bigloo library for the host-platform. At the same time one might want to build the Bigloo-executable (for the host-platform) too, but this is not a requirement.

Programs can then be cross-compiled simply by telling Bigloo to use the host-library.

Note: if the cross-compiled executable uses shared libraries, then Bigloo's cross-compiled libraries have to be copied to the host platform. Static executables are self-contained and can be run without modification on the host.

32.2 Building the Bigloo library for the host-platform

We assume that we have a C cross-compiler `CC` and an empty Bigloo source tree. As a first step the configuration script must be executed. However, Bigloo's configure script requires some runtime-information from the host and thus needs access to the host-machine. This is accomplished by passing a hostsh-script to configure.

32.2.1 Hostsh

A *hostsh* script is passed to Bigloo's configuration script and is invoked whenever a command should be executed on the host-side.

There are already three example scripts inside Bigloo's source tree that are located in the `examples/hostsh` directory.

- `by-hand/kdiallog-host.sh` asks (using KDE's kdialog) to execute the command by hand on the host-side and to report the result.
- `ssh/ssh-copy.sh` copies the file by `ssh` and executes it then on the other side. If you intend to use that file, you must edit it for setting the variables describing the remote ssh connection. For the variables `HOST`, `SSH_PORT`, and `USER` must be modified. Note: there exists an Ipod/Iphone version that automatically signs on jail-broken devices.

- and finally, as last resort, there exists a `netcat` version if no `ssh` is available. This one can be used on devices that only have telnet access, and where ssh is not available. Its only requirement is a running netcat on the host-side (which should be easily achievable since there exists a working cross compiler).

32.2.2 Building

Armed with a working cross-compiler `CC` and a script `HOSTSH` that invokes commands and executables on the host side the configure invocation is simple:

```
./configure \
  –prefix=[PREFIX_PATH_ON_TARGET] \
  –cc=[CC] \
  –hostsh=[HOSTSH] \
  –thirdparty-configure-opt=[options]
```

Other configuration options are of course possible too.

For instance, for configuring Bigloo for a Raspberry model 2.

```
./configure \
  –cc=/usr/bin/arm-linux-gnueabi-gcc-6 \
  –hostsh=$PWD/ssh-copy.sh \
  –thirdparty-configure-opt=–host=arm-linux-gnueabi
```

Once the configuration has finished one can build Bigloo (and its library) simply by calling `make`. This will build the libraries as well as the binaries.

If shared libraries are needed on the host platform one still needs to install them. The easiest way is probably to install them temporary on a build system inside a special directory and then copy them from there to the host system.

```
make DESTDIR=[temporary-directory] install
```

Only the `lib` directory is needed on the host side.

32.3 Cross Compiling Bigloo Programs

Once the host-library exists cross compilation is straightforward. Using the `-lib-dir` compilation flag one simply has to pass the library-directory to Bigloo.

```
bigloo -lib-dir [path-to-cross-compiled-library] ....
```

Bigloo will automatically use the same C cross-compiler and compilation flags that have been used to build the library.

32.4 Caveats

In general Bigloo's cross-compilation works fine, but developers should be aware of some limitations:

- Macros will be executed on the build platform. The macro-environment (and in particular its integer types) might not be the same. For instance an `elong` on the build-system might be of different size than an `elong` on the host-system.
- Bigloo will read numbers on the build system and adapt the container size accordingly. Suppose for instance that the build system features 64bit longs, but the host system only allows for 32bit longs. The number `2^35` fits easily into a long on the build-system

but will overflow on the host-system. The container will however be determined on the build system and thus a long will be used. This is only a problem for big integer literals.

- A cross-compiled Bigloo uses (by default) the same C compiler that has been used to compile the Bigloo. Once the executable has been transferred to the host-system the C cross-compiler does very likely not exist anymore. Therefore Bigloo will need to be invoked with the `-cc` flag on the host-system (under the assumption that there exists a C compiler).

  This drawback can be eliminated by directly compiling Bigloo on the host (since there exists a C compiler).

32.5 Examples

In this example we will show how to compile for a host-machine that has ssh-access.

We assume

- a working Bigloo (should be the same version as the one that is going to be compiled for the host) in the PATH.
- ssh access to the host. This access should be without password (using keys). The system should be accessible by `ssh [host]` (where `[host]` should be replaced with the correct address).
- a C cross-compiler `CC` running on the build-system and compiling for the host.

With these preconditions satisfied we can first build Bigloo for the host-system:

```
$ ./configure –hostsh="$PWD/examples/hostsh/ssh/ssh-copy.sh [host]" –cc=[CC]
$ make
$ make DESTDIR=[TMP] install
```

Now let's compile a simple hello-world for the host.

```
$ cat > /tmp/hello.scm <<EOF
(module hello (main main))
(define (main args) (print "hello world"))
EOF

$ bigloo -static-all-bigloo -lib-dir [TMP]/lib/3.2c/ -o /tmp/hello /tmp/hello.scm
```

The generated executable should be able to run on the host.

33 User Extensions

The extension package system allows the language compiled by Bigloo to be extended and this is achieved by associating an *extension file* with a suffix. The *extension file* is loaded at the beginning of a compilation and it can do three things: call extern programs (unix programs); define macros; modify the values of some of the compiler's variables (for example, the list of the libraries to be linked with). The Bigloo's initializing procedure is the following:

- If it exists, Bigloo loads the runtime-command file, see Section Chapter 31 [Compiler Description], page 275.
- It then parses the command line to find the source file to compile.
- It extracts the source file suffix and looks it up in its `*auto-mode*` variable.
- If the suffix is found, the associated file is loaded. This file could contain a function named `*extend-entry*` which must accept a list as argument. It is invoked with the Bigloo's unparsed arguments.
- The result of the `*extend-entry*` application has to be a regular list of arguments and these are parsed by Bigloo.

For now, two extension packages exist: the Meroon package which is a native version of the Christian Queinnec object language; the Camloo [SerranoWeis94] package which is a front end compiler for the Caml language [Caml-light]

Furthermore, Bigloo supports the `-extend` option which forces the usage of an extension file. When Bigloo encounters this option, it immediately loads the extension file, invoking the function `*extend-entry*` with the list of arguments which have not been parsed yet.

The extension files are always sought in the directory containing the Bigloo's libraries.

33.1 User pass

Bigloo allows the user to add a special pass to the regular compilation, this pass taking place *before* macro expansion. There are two ways to add a user pass.

- Add a compiled pass: The module `user_user` (in the "comptime/User/user.scm" file) is the user entry pass point. To add a compiled pass, put the code of the pass in this directory, import your new modules in `user_user` and modify the `user-walk` function.
- Add an interpreted pass: Set the value of `*user-pass*`, which has to be a unary function, in your `.bigloorc` file and Bigloo will invoke it with the code as argument.

34 Bigloo Development Environment

Bigloo release 2.0 or more recent contains an Integrated Development Environment. This environment proposes some services:

- Automatic Makefile creation and update.
- Code browsing.
- Revision control.
- Symbol debugging.
- Profiling.
- On-line documentation.
- Source code interpretation.
- Source code expansion.
- Literate programming.

The environment relies on Bigloo tools:

- `bglafile`: a Module access file generator (see Section 2.6 [Module Access File], page 16).
- `bglmake`: a Makefile creator.
- `bgldepend`: a that creates Bigloo dependencies in makefiles.
- `bglpp`: a pretty printer.
- `bglprof`: a profiler
- `bgltags`: a generator of tag file for Emacs.

Each of these tools comes with a dedicated manual page and are not described in this documentation.

Extra tools are required for the environment to go its full speed:

- `Emacs` version 21 (or more recent) (`http://www.emacs.org/`) or `Xemacs` version 20.4 (or more recent) (`http://www.xemacs.org/`).
- `prcs` version 1.2.1 or more recent (`http://prcs.sourceforge.net/`).
- `gdb` version 4.17 or more recent (`http://www.cygnus.com/gdb/`).
- `gprof` (`ftp:/prep.ai.mit.edu:pub/gnu/`).
- `texinfo` (`ftp:/prep.ai.mit.edu:pub/gnu/`).
- `gmake` (`ftp:/prep.ai.mit.edu:pub/gnu/`).

The following document describes the Bee, the Bigloo Emacs Environment.

34.1 Installing the Bee

The standard procedure for installing Bigloo handles the compilation of all tools required by the Bee. Additional Emacs-lisp code have to be appended to your `.emacs` file:

```
(autoload 'bdb "bdb" "bdb mode" t)
(autoload 'bee-mode "bee-mode" "bee mode" t)

(setq auto-mode-alist
      (append '(("\\.scm$" . bee-mode)
```

```
   ("\\.sch$" . bee-mode)
   ("\\.scme$" . bee-mode)
   ("\\.bgl$" . bee-mode)
   ("\\.bee$" . bee-mode))
 auto-mode-alist))
```

This code will force `emacs` to switch to Bee mode when editing Scheme source files.

34.2 Entering the Bee

Once, your `.emacs` is updated, to start the Bee you just need to edit a file suffixed with one of the suffix listed in *auto-mode-alist* with Emacs. You may either enter the Bee within Emacs with `ESC-X: bee-mode`.

34.3 The *Bee Root Directory*

The *Bee Root Directory* is the directory that contains information files that describe a project. When editing a file, the Bee tries to automatically setup the *Bee Root Directory*. For that, it seeks one of the following file: `Makefile`, `.afile` or `.etags`. This search unwind directories until the root directory is reached or until the number of scanned directories is more than the value of the list variable `bee-root-search-depth`.

An alternative *Bee Root Directory* may be set. This is done clicking on the `Root` of the tool bar icon.

34.4 Building a Makefile

Once, the *Bee Root Directory* has been setup (it is printed on the left part to the Emacs modeline), a `Makefile` can be automatically produced. It can be achieved clicking on the `Mkmf` icon of the tool bar, using the popup menu (*`button-3`*) entries, or using one of the two keyboard bindings *`C-c C-c C-a`* or *`C-c C-c C-l`*. When creating a `Makefile`, you will be asked to give a file name. This file must be the one that is the main entry point of your program or the one that implements an library heap file.

When the `Makefile` already exists, using the same bindings update Makefile, re-generate `.afile` and `.etags` files.

34.5 Compiling

Once a `Makefile` exists, it is possible to compile a program (or a library). Use either the tool bar icon `Compile`, the popup menu entry or *`C-c C-c C-c`*. If no `Makefile` exists, the Bee will emit a single file compilation.

34.6 Interpreting

Scheme source code may be interpreted within the Bee instead of been compiled prior to be executed. This facility could be convenient for fast prototyping. A *Read eval print* loop (henceforth *Repl*) could be spawned using the `Repl` icon of the tool bar, using the popup menu entry or using the *`C-c C-r C-r`* binding.

Parts or the whole buffer may be sent to *repl*.

- *`C-c C-r b`* sends the whole buffer.

- `C-c C-r d` sends the define form the cursor is in.
- `C-c C-r l` sends the s-expression that preceeds the cursor.
- `C-c C-r t` sends the top level s-expression the cursor is in.
- `C-c C-r r` sends the marked region.

34.7 Pretty Printing

The whole buffer may be pretty printed (long source lines are split) using the `Lisp` icon of the tool bar, using the popup menu entry of using `C-c C-i tab`.

Parts or the buffer may be indented (no line is split).

- `C-c C-i d` indents the define form the cursor is in.
- `C-c C-i l` indents the s-expression that preceeds the cursor.
- `C-c C-i t` indents the top level s-expression the cursor is in.

34.8 Expansing

For debug purposes, result of the source code macro expansion may be checked within the Bee.

Parts or the whole buffer may be sent to *repl*.

- `C-c C-e C-e` expands the whole buffer.
- `C-c C-e C-d` expands the define form the cursor is in.
- `C-c C-e C-l` expands the s-expression that preceeds the cursor.
- `C-c C-e C-t` expands the top level s-expression the cursor is in.
- `C-c C-e C-r` expands the marked region.

When a part of the buffer is expanded (by opposition to the whole buffer), the buffer is scan for macro definitions. These macros will be used for expanding the requested form.

34.9 On-line Documentation

On-line documentation may be popped up. This is always done, clicking on the `Info` icon of the tool bar or `C-c C-d i`. If an emacs region is active, the documentation about that region will be popped up. If the cursor is at a Scheme identifier, the documentation of that identifier will be printed. Otherwise, the user will be prompted for the Section of the documentation to be printed.

Clicking on the `?` icon tool bar, pops up a short description of the Bigloo compiler options.

The Bee uses `info` files for printing On-line documentation. It always search the standard documentation and the standard definition of Scheme. It is possible to add extra `info` files to be searched. The Bee always checks for a directory `info` in the *Bee Root Directory*. If such a directory exists, contained file will be considered for the search of a document.

34.10 Searching for Source Code

Searching for source (variable declaration, module definition, variable usage) is supported by the Bee. Clicking on the `Find` icon of the tool bar will pops up the definition of the variable the cursor is in or the definition of the module the cursor is in. These two operations may be requested using *C-x 5 .* for searching a variable definition, *C-c C-d m* for a module definition.

Information and usages of a variable may be printed using either the `Doc` icon of the tool bar or the *C-c C-d u* key binding.

34.11 Importing and Exporting

Bigloo bindings (functions and variables) may be automatically inserted in an export module clause (see Chapter 2 [Modules], page 7). Bring the cursor to an identifier of a binding that has to be exported then, either click on the `Export` tool bar icon or use the *C-c C-m b* key binding.

Bigloo bindings may be automatically inserted in an import module clause. Bring the cursor to an identifier of a binding that has to be imported. Either click on the `Import` tool bar icon or use the *C-c C-m i* key binding. The Bee, will search the modules for the wanted binding.

Foreign bindings (e.g. C variables and C functions) may be automatically inserted in the file module clause. Click on the `Extern` tool bar icon or use the key binding *C-c C-m c* to import whole the definition of an extern file. You will, be prompted an extern file name to be imported. This operation *automatically updates* the `Makefile` for reflecting that the extern file is required in the compilation.

34.12 Debugging

The Bigloo symbolic Debugger may be spawned either clicking on the `Bdb` tool bar icon or using the key binding *C-c C-b C-b*. Once the debugger is not is possible to connect the current buffer to the debugger. This is done using the tool bar icon `Connect` or the key binding *C-c C-b c*. This enables breakpoints to be inserted using mouse clicks.

34.13 Profiling

Automatically produced `Makefile` provides entry for profiling. In order to get a profile you must first compile your application for profiling. This is done using a popup menu entry or the *C-c C-p c* key binding. Once your program compiled you can run for profile using a popup menu entry of the *C-c C-p r* key binding. This last will run your program, run `bglprof` to get the profile and this will pops up a window displaying the profile informations.

34.14 Revision Control

Submitting a new revision is done using *C-c C-v i* or using an menu bar entry. This builds an new revision for the entire project. The file that compose the project are listed in the `pop` entry of the *Bee Root Directory* `Makefile`.

Checking out an older version of the file currently edited is done using the key binding `C-c C-v C-o`. This is not a retrieval of the entire project. Global check out may be performed manually.

Comparing the version of the file currently edited with older one is done using `C-c C-v d`. A *diff* of the two buffers will be popped up.

With both checking out and comparison of versions. A window presenting all the available version will be popped up to let you choose which version you would like to inspect.

34.15 Literate Programming

The Bee does not provide real *Literate Programming*. The reason is that we think that when editing documentation we want to benefit the full power of context-sensitive editors and we don't want to edit the documentation is the same editor *mode* as the one we use when editing source code. Nevertheless it is possible to place anchors within the source file to the corresponding documentation file. Then, by the means of simple mouse clicks, it becomes possible to edit the documentation of peace of codes. The current Bee literate programming system only supports the *Texinfo* file format.

For that purpose three anchors are available: `path`, `node` and `deffn`. All anchor have to be delimited with `@` characters.

- `path`: this anchor set the path to the file containing the documentation. Thus,

```
(module foo
   ;; @path manuals/foo.texi@
   ...)
```

  Tells the Bee that the documentation for the module `foo` is located in the file named `manuals/foo.texi`.

- `node`: sets the name of the node that documents this particular source file code.

```
(module foo
   ;; @path manuals/foo.texi@
   ;; @node Foo@
   ...)
```

- `deffn`: each variable binding may point to its documentation. For that, it suffices to use the `deffn` anchor just before the variable definition or within the s-expression that defines the variable.

```
;; @deffn foo@
(define (foo . chars)
   ...)
```

  or

```
(define (foo . chars)
   ;; @deffn foo@
   ...)
```

  When clicking on that anchor, the Bee will search the documentation file named by the `path` anchor and within that file, will search for a *texinfo* `deffn` command that defines the variable named in the anchor.

35 Global Index

#

#! Unix shell interpreter 272
#; 17
#| 17
#a<ddd> 38
#b 32
#d 32
#ex 32
#l 32
#lx 32
#o 32
#x 32
#z 32
#zx 32

&

&error 175
&eval-warning 176
&exception 175
&http-error 176
&http-redirection 176
&http-redirection-error 176
&http-status-error 176
`&imap-error` 216
`&imap-parse-error` 216
&io-closed-error 176
&io-error 175
&io-file-not-found-error 176
&io-malformed-url-error 80, 176
&io-parse-error 176
&io-port-error 175
&io-read-error 176
&io-unknown-host-error 176
&io-write-error 176
`&mailbox-error` 214
`&maildir-error` 217
&process-exception 176
&type-error 175
&warning 176

'

'datum 18

*

`*` 34
`*auto-mode*` 303
`*bx` 34
`*dynamic-load-path*` 226
`*elong` 34
`*extend-entry*` 303
`*fl` 34
`*fx` 34
`*llong` 34
`*load-path*` 16, 225
`*pp-case*` 65
`*pp-width*` 65
`*user-pass*` 303

+

`+` 34
`+bx` 34
`+elong` 34
`+fl` 34
`+fx` 34
`+llong` 34

-

`-` 34
`->` 120
`-bx` 34
`-elong` 34
`-fl` 34
`-fx` 34
-jvm-cinit-module 253
`-llong` 34

.

`.afile` 16
`.bigloorc` 277
.jfile 247

/

`/` 35
`/bx` 35
`/elong` 35
`/fl` 35
`/fx` 35
`/llong` 35

<

< 33
<= 34
<=bx 34
<=elong 34
<=fl 34
<=fx 34
<=llong 34
<bx 34
<constant> 18
<elong 34
<fl 34
<fx 34
<lllong 34
<variable> 18

=

= 33
=bx 33
=elong 33
=fl 33
=fx 33
=llong 33

>

> 34
>= 34
>=bx 34
>=elong 34
>=fl 34
>=fx 34
>=llong 34
>bx 34
>elong 34
>fl 34
>fx 34
>lllong 34

‘

‘ 22

@

@ 15

8

8bits->utf8 46
8bits->utf8! 46

A

a simple example of Lalr(1) parsing 140
A complete library example 262
A new way of reading 129
abandoned-mutex-exception? 193
abs 35
absfl 35
abstract-class 117
Acknowledgements 1
acos 36
acosfl 36
aes-ctr-decrypt 158
aes-ctr-decrypt-file 158
aes-ctr-decrypt-mmap 158
aes-ctr-decrypt-port 158
aes-ctr-decrypt-string 158
aes-ctr-encrypt 158
aes-ctr-encrypt-file 158
aes-ctr-encrypt-mmap 158
aes-ctr-encrypt-port 158
aes-ctr-encrypt-string 158
and 19
and-let* 19
any 28
append 27
append! 27
append-map 50
append-map! 50
append-output-binary-file 70
append-output-file 59
apply 50
args-parse 153
args-parse-usage 153
arguments parsing 153
ascii-string? 45
asin 36
asinfl 36
assert 177
assertions 173
assoc 28
assq 28
assv 28
asynchronous signal 193
atan 36
atanfl 36
automatic extern clauses generation 236
Automatic Java clauses generation 249

B

base64 ... 98
base64-decode ... 98
base64-decode-port ... 98
base64-encode ... 98
base64-encode-port ... 98
basename ... 79
begin ... 21
bglafile ... 16
bibtex ... 219
bibtex-file ... 219
bibtex-parse-authors ... 219
bibtex-port ... 219
bibtex-string ... 219
bigloo development environment ... 305
bigloo variable ... 221, 222
bigloo-class-demangle ... 245
bigloo-class-mangled? ... 245
bigloo-compiler-debug ... 231
bigloo-compiler-debug-set! ... 231
bigloo-config ... 76
bigloo-debug ... 231
bigloo-debug-set! ... 231
bigloo-demangle ... 245
bigloo-dns-cache-validity-timeout ... 232
bigloo-dns-cache-validity-timeout-set! ... 232
bigloo-dns-enable-cache ... 232
bigloo-dns-enable-cache-set! ... 232
bigloo-eval-strict-module ... 223, 232
bigloo-eval-strict-module-set! ... 232
bigloo-library-path ... 258
bigloo-library-path-set! ... 258
bigloo-mangle ... 245
bigloo-mangled? ... 245
bigloo-module-mangle ... 245
bigloo-need-mangling ... 245
bigloo-strict-r5rs-strings ... 231
bigloo-strict-r5rs-strings-set! ... 231
bigloo-trace ... 231
bigloo-trace-color ... 231
bigloo-trace-color-set! ... 231
bigloo-trace-set! ... 231
bigloo-trace-stack-depth ... 232
bigloo-trace-stack-depth-set! ... 232
bigloo-warning ... 231
bigloo-warning-set! ... 231
bignum ... 32
bignum->octet-string ... 36
bignum->string ... 36
bignum? ... 32
binary-port? ... 70
bind-exit ... 51
bit manipulation ... 72
bit-and ... 72
bit-andelong ... 72
bit-andllong ... 72
bit-lsh ... 73
bit-lshelong ... 73
bit-lshllong ... 73
bit-not ... 72
bit-notelong ... 72
bit-notllong ... 73
bit-or ... 72
bit-orelong ... 72
bit-orllong ... 72
bit-rsh ... 73
bit-rshelong ... 73
bit-rshllong ... 73
bit-ursh ... 73
bit-urshelong ... 73
bit-urshllong ... 73
bit-xor ... 72
bit-xorelong ... 72
bit-xorllong ... 72
blit-string! ... 43
bm-mmap ... 110
bm-string ... 110
bm-table ... 110
bmh-mmap ... 111
bmh-string ... 111
bmh-table ... 111
boolean? ... 25
Booleans ... 25
Boyer, Moore ... 110
Boyer, Moore, Horspool ... 110
broadcast! ... 192
Building a library ... 259
building a makefile ... 306
byte-code-compile ... 223
byte-code-run ... 223

C

C arrays ... 241
C atomic types ... 237
C enum ... 242
C functions ... 242
C interface ... 235
C null pointers ... 240
C opaque ... 243
C pointers ... 239
C requirement ... 275
C structure and union types ... 238
caaar ... 27
caadr ... 27
caar ... 27
cadar ... 27
caddr ... 27
cadr ... 27
Calendar ... 95
call ... 18
call-next-method ... 121
call-with-append-file ... 54
call-with-input-file ... 54
call-with-input-string ... 54
call-with-output-file ... 54

call-with-output-string 54
call-with-values 53
call/cc 51, 52
car 27
case 19
cdddar 27
cddddr 27
cdr 27
ceiling 35
ceilingfl 35
certificate-issuer 95
certificate-subject 95
Certificates 95
char->integer 38
char->ucs2 39
char-alphabetic? 38
char-ci<=? 38
char-ci<? 38
char-ci=? 38
char-ci>=? 38
char-ci>? 38
char-downcase 38
char-lower-case? 38
char-numeric? 38
char-ready? 61
char-ready? and run-process 61
char-upcase 38
char-upper-case? 38
char-whitespace? 38
char<=? 38
char<? 38
char=? 38
char>=? 38
char>? 38
char? 38
characters 38
chdir 80
chmod 83
cigloo 236
circular representation 65
class 117
Class declaration 117
class-all-fields 126
class-creator 127
class-field-accessor 126
class-field-info 127
class-field-mutable? 127
class-field-mutator 127
class-field-name 126
class-field? 126
class-fields 126
class-name 125
class-nil 119
class-predicate 128
class-subclasses 125
class-super 125
class? 125
client-server 90
client-socket-use-ssl! 94
close-binary-port 70
close-input-port 59
close-mmap 67
close-output-port 59
close-process-ports 85
close-semaphore 196
closed-input-port? 59
closed-output-port? 59
closelog 79
co-instantiate 120
cobj 238
Color 210
command line parsing 153
command-line 77
comments 17
Compiler description 275
compiler JVM options 247
compiler options 277
Compiling 306
compiling and linking with a library 255
Compiling with the JVM back-end 247
Complete-DSA-Key 163
Complete-ElGamal-Key 164
Complete-RSA-Key 160
complex? 32
cond 19
cond-expand 269
condition-variable-broadcast! 184
condition-variable-name 184
condition-variable-signal! 184
condition-variable-specific 184
condition-variable-specific-set! 184
condition-variable-wait! 184
condition-variable? 184
Connecting Scheme and Java code 249
cons 27
cons* 28
control features 50
copy-file 82
copy-vector 48
cos 35
cosfl 36
cp1252->utf8 46
cp1252->utf8! 46
cram-md5sum-string 98
crc 100
crc-elong-le::elong 101
crc-elong::elong 101
crc-file 100
crc-length 100
crc-llong-le::llong 101
crc-llong::llong 101
crc-long-le::long 101
crc-long::long 101
crc-mmap 100
crc-names 100
crc-polynomial 100

crc-polynomial-be->le 102
crc-polynomial-le 100
crc-port 100
crc-string 100
CRC 99
create-hashtable 74
Creating and accessing objects 119
cross compilation 299
cryptography 155
Csv 221
csv-for-each 221
csv-map 221
current-date 96
current-error-port 55
current-exception-handler 174
current-input-port 55
current-microseconds 96
current-nanoseconds 96
current-output-port 55
current-scheduler 191
current-seconds 96
current-thread 181
current-time 193
Custom types 267

D

database 199
datagram-socket-receive 91
datagram-socket-send 91
Date 95
date 77
date->iso8601-date 98
date->nanoseconds 96
date->rfc2822-date 97
date->seconds 96
date->string 96
date->utc-string 96
date-copy 96
date-day 96
date-hour 96
date-is-dst 97
date-minute 96
date-month 97
date-month-length 97
date-nanosecond 96
date-second 96
date-timezone 97
date-wday 97
date-week-day 97
date-yday 97
date-year 97
date-year-day 97
date? 95
day-aname 97
day-name 97
day-seconds 97
debug 179
debugging 308
Debugging Lalr Grammars 139
declare-library! 256
Declaring abstract Java classes 250
Declaring Java arrays 250
Declaring Java classes 249
decrypt-file::bstring 158
decrypt-mmap::bstring 158
decrypt-port::bstring 158
decrypt-sendchars 158
decrypt-string::bstring 158
decrypt::bstring 158
default-scheduler 191
define 22, 233
define-expander 229
define-generic 121
define-inline 15, 233
define-macro 229
define-method 121
define-reader-ctor 60
define-record-type 114
define-struct 113
define-syntax 229
defining an extern type 237
Definitions 22
delay 22
delete 28
delete! 28
delete-directory 82
delete-duplicates 30
delete-duplicates! 30
delete-file 81
delete-semaphore 196
Digest 98
directives 8
directory->list 82
directory->path-list 82
directory? 82
dirname 80
display 63
display* 63
display-circle 65
display-string 65
display-substring 65
do 22
double->ieee-string 37
double->llong-bits 37
drop 28
dsa-sign 163
dsa-verify 164
DSA-Key 163
DSSSL formal argument lists 23
DSSSL support 23
dump-trace-stack 173
duplicate 120
duplicate::*class* 120
dynamic-load 225
dynamic-load-symbol 227

dynamic-load-symbol-get ... 227
dynamic-load-symbol-set ... 227
dynamic-unload ... 226
dynamic-wind ... 52

E

Efficiency ... 275
elgamal-decrypt ... 164
elgamal-encrypt ... 164
ElGamal-Key ... 164
elgamal-key-length ... 164
elong->fixnum ... 37
elong->flonum ... 37
elong->string ... 36
elong? ... 32
email-normalize ... 213
Embedded Bigloo applications ... 245
encrypt-file::bstring ... 155
encrypt-mmap::bstring ... 155
encrypt-port::bstring ... 155
encrypt-sendchars ... 156
encrypt-string::bstring ... 155
encrypt::bstring ... 155
Entering the Bee ... 306
eof-object? ... 61
eq? ... 25
equal? ... 27
Equality ... 126
Equivalence predicates ... 25
eqv? ... 25
error ... 173
error handling ... 173
error-notify ... 174
error/location ... 173
eval ... 12, 223
Eval and the foreign interface ... 227
Eval command line options ... 227
Eval operator inlining ... 223
Eval standard functions ... 223
even? ... 33
evenbx? ... 33
evenelong? ... 33
evenfl? ... 33
evenfx? ... 33
evenllong? ... 33
every ... 28
exact fixnum ... 32
exact->inexact ... 36
exact? ... 33
examples of regular grammar ... 135
exception-notify ... 174
exceptions ... 174
executable-name ... 77
exif ... 203
exif-date->date ... 204
exit ... 77
exp ... 35
expand ... 224
expand-once ... 224
Expansing ... 307
expansion passing style ... 229
expfl ... 35
explicit typing ... 233
export ... 9
exporting a Scheme variable ... 237
Exporting Scheme variables to Java ... 251
Expressions ... 17
expt ... 36
exptfl ... 36
extern ... 12
extract-private-rsa-key ... 160
extract-public-dsa-key ... 163
extract-public-elgamal-key ... 164
extract-public-rsa-key ... 160

F

Fair Threads ... 185, 196
fast string search ... 109
file handling ... 79
file->string ... 63
file-access-time ... 82
file-change-time ... 82
file-exists? ... 81
file-gid ... 83
file-gzip? ... 81
file-mode ... 83
file-modification-time ... 82
file-musictag ... 204
file-name->list ... 80
file-name-canonicalize ... 80
file-name-canonicalize! ... 80
file-name-unix-canonicalize ... 80
file-name-unix-canonicalize! ... 80
file-separator ... 78
file-size ... 83
file-times-set! ... 82
file-type ... 83
file-uid ... 83
filter ... 50
filter! ... 50
filter-map ... 51
final-class ... 117
find ... 29
find-class ... 125
find-class-field ... 126
find-file/path ... 81
find-tail ... 29
finitefl? ... 36
fixnum ... 32
fixnum (long) ... 32
fixnum->elong ... 37
fixnum->flonum ... 37
fixnum->llong ... 37
fixnum? ... 32

flac-musictag ... 204
float->ieee-string ... 37
float->int-bits ... 37
flonum ... 32
flonum->elong ... 37
flonum->fixnum ... 37
flonum->llong ... 37
flonum? ... 32
floor ... 35
floorfl ... 35
flush-binary-port ... 70
flush-output-port ... 64
for-each ... 50
force ... 51
format ... 64
fprint ... 64
fprintf ... 65
free-pragma::*ident* ... 244
from ... 10

G

gb2312->ucs2 ... 220
gcd ... 35
generate-rsa-key ... 160
Generic functions ... 121
Genericity of arithmetic procedures ... 276
gensym ... 31
genuuid ... 31
get-class-serialization ... 72
get-custom-serialization ... 72
get-interfaces ... 91
get-opaque-serialization ... 72
get-output-string ... 59
get-procedure-serialization ... 72
get-process-serialization ... 72
get-prompter ... 224
get-protocol ... 91
get-protocols ... 91
get-repl-error-notifier ... 224
get-signal-handler ... 77
get-trace-stack ... 173
getenv ... 77
getgid ... 78
getgroups ... 79
getpid ... 78
getppid ... 79
getprop ... 31
getpwnam ... 79
getpwuid ... 79
getuid ... 78
gunzip ... 68
gunzip-parse-header ... 69
gunzip-sendchars ... 68
gzip ... 57, 68

H

hardware fixnum ... 32
hashtable->list ... 75
hashtable->vector ... 75
hashtable-add! ... 75
hashtable-clear! ... 76
hashtable-contains? ... 75
hashtable-filter! ... 76
hashtable-for-each ... 75
hashtable-get ... 75
hashtable-key-list ... 75
hashtable-map ... 75
hashtable-put! ... 75
hashtable-remove! ... 75
hashtable-size ... 74
hashtable-update! ... 75
hashtable-weak-data? ... 74
hashtable-weak-keys? ... 74
hashtable? ... 74
hmac-md5sum-string ... 98
hmac-sha1sum-string ... 99
hmac-sha256sum-string ... 99
homogeneous vectors ... 49
host ... 91
hostinfo ... 91
hostname ... 78
hsl->rgb ... 210
hsv->rgb ... 210
http ... 102, 103
http-chunks->port ... 105
http-chunks->procedure ... 105
http-parse-header ... 105
http-parse-response ... 105
http-parse-status-line ... 104
http-read-crlf ... 104
http-read-line ... 104
http-response-body->port ... 105
http-send-chunks ... 105
http-url-parse ... 102
hyphenate ... 219

I

id3, m3u ... 204
id3::musictag ... 205
ieee-string->double ... 37
ieee-string->float ... 37
if ... 19
ignore ... 134
imap ... 214, 216
imap-capability ... 217
imap-login ... 216
imap-logout ... 217
import ... 8
importing an extern function ... 236
importing an extern variable ... 236
Importing and Exporting ... 308
include ... 8

including an extern file ... 236
inexact->exact ... 36
inexact? ... 33
infinitefl? ... 36
inflate-sendchars ... 68
inline procedure ... 15
input and output ... 54
input-char ... 71
input-fill-string! ... 71
input-obj ... 70
input-port-close-hook ... 55
input-port-close-hook-set! ... 55
input-port-length ... 54
input-port-name ... 54, 59
input-port-name-set! ... 54
input-port-position ... 60
input-port-reopen! ... 55, 60
input-port-timeout-set! ... 54
input-port? ... 54
input-string ... 71
input-string-port? ... 54
installing the bee ... 305
instantiate ... 119
instantiate::*class* ... 119
instantiate::fthread ... 186
instantiate::pthread ... 194
int-bits->float ... 38
integer->char ... 38
integer->second ... 97
integer->string ... 36
integer->string/padding ... 36
integer->ucs2 ... 39
integer? ... 32
interaction-environment ... 223
internet ... 102
Interpreting ... 306
Introspection ... 126
inverse-utf8-table ... 47
io-unknown-host-error ... 91
iota ... 30
IP number ... 87
is-a? ... 125
isa? ... 119
iso-latin->utf8 ... 46
iso-latin->utf8! ... 46
iso8601-date->date ... 98
iso8601-parse-date ... 98

J

java ... 12
Java interface ... 247
jfile ... 247
jigloo ... 249
join-timeout-exception? ... 193
jpeg-exif ... 203
jpeg-exif-comment-set! ... 203
JVM requirement ... 275

K

keyword->string ... 32
keyword->symbol ... 32
keyword? ... 32
keywords ... 31
kmp-mmap ... 109
kmp-string ... 109
kmp-table ... 109
Knuth, Morris, and Pratt ... 109

L

labels ... 21, 233
Lalr grammar and Regular grammar ... 139
Lalr parsing ... 137
Lalr precedence and associativity ... 138
lalr(1) grammar definition ... 137
lalr-grammar ... 137
lambda ... 18
last-pair ... 28
lcm ... 35
leap-year? ... 97
length ... 27
let ... 20, 233
let* ... 20, 233
let-syntax ... 229
letrec ... 20, 233
letrec* ... 21
letrec-syntax ... 229
libraries ... 255
library ... 13
library and eval ... 258
Library and inline functions ... 257
Library and macros ... 262
Library and modules ... 261
library and repl ... 259
library-exists? ... 258
library-load ... 258
library-translation-table-add! ... 257
Limitation of the JVM back-end ... 248
linking ... 275
list ... 27
list->string ... 42
list->TAGvector ... 50
list->ucs2-string ... 45
list->vector ... 47
list-copy ... 30
list-ref ... 28
list-split ... 29
list-tabulate ... 29
list-tail ... 28
list? ... 27
literate programming ... 309
llong->fixnum ... 37
llong->flonum ... 37
llong->string ... 36
llong-bits->double ... 38
llong? ... 32

load ... 11, 225
load-hyphens ... 219
loada ... 225
loadq ... 225
lockf ... 67
log ... 35
logfl ... 35
long fixnum ... 32

M

macro expansion ... 229
mail ... 211
mail-header->list ... 213
mailbox ... 214
mailbox-close ... 214
mailbox-folder-create! ... 214
mailbox-folder-dates ... 215
mailbox-folder-delete! ... 215
mailbox-folder-delete-messages! ... 215
mailbox-folder-exists? ... 215
mailbox-folder-header-fields ... 215
mailbox-folder-move! ... 215
mailbox-folder-rename! ... 215
mailbox-folder-select! ... 214
mailbox-folder-status ... 215
mailbox-folder-uids ... 215
mailbox-folder-unselect! ... 214
mailbox-folders ... 214
mailbox-hostname ... 214
mailbox-message ... 215
mailbox-message-body ... 215
mailbox-message-create! ... 216
mailbox-message-delete! ... 216
mailbox-message-flags ... 216
mailbox-message-flags-set! ... 216
mailbox-message-header ... 215
mailbox-message-header-field ... 216
mailbox-message-header-list ... 215
mailbox-message-info ... 216
mailbox-message-move! ... 216
mailbox-message-path ... 215
mailbox-message-size ... 216
mailbox-prefix ... 214
mailbox-separator ... 214
mailbox-subscribe! ... 215
mailbox-unsubscribe! ... 215
maildir ... 214, 217
main ... 8
make-asynchronous-signal ... 193
make-client-socket ... 86
make-condition-variable ... 184
make-csv-lexer ... 221
make-datagram-server-socket ... 90
make-datagram-unbound-socket ... 90
make-date ... 96
make-directories ... 82
make-directory ... 82
make-elong ... 32
make-file-name ... 80
make-file-path ... 80
make-hashtable ... 74
make-hyphens ... 220
make-list ... 29
make-llong ... 32
make-mutex ... 182
make-scheduler ... 191
make-server-socket ... 88
make-shared-library-name ... 81
make-spinlock ... 182
make-ssl-client-socket ... 92
make-ssl-server-socket ... 94
make-static-library-name ... 81
make-string ... 39
make-string-ptr-null ... 240
make-symlink ... 82
make-TAGvector ... 50
make-thread ... 194
make-ucs2-string ... 44
make-vector ... 47
make-void*-null ... 240
make-weakptr ... 73
map ... 50
map! ... 50
match-case ... 107
match-lambda ... 107
max ... 33
maxbx ... 33
maxfl ... 33
maxfx ... 33
maxvalelong ... 32
maxvalfx ... 32
maxvalllong ... 32
md5 ... 98
md5sum ... 98
md5sum-file ... 98
md5sum-mmap ... 98
md5sum-port ... 98
md5sum-string ... 98
member ... 28
memq ... 28
memv ... 28
mime ... 211
mime-content-decode ... 211
mime-content-decode-port ... 211
mime-content-disposition ... 211
mime-content-disposition-decode ... 211
mime-content-disposition-decode-port ... 211
mime-content-type ... 211
mime-multipart ... 211
mime-multipart-decode ... 212
mime-multipart-decode-port ... 212
min ... 33
minbx ... 33
minfl ... 33
minfx ... 33

minvalelong ... 32
minvalfx ... 32
minvalllong ... 32
mixer ... 205
mixer-close ... 205
mixer-volume-get ... 205
mixer-volume-set! ... 205
mmap ... 67
mmap-get-char ... 68
mmap-get-string ... 68
mmap-length ... 67
mmap-name ... 67
mmap-put-char! ... 68
mmap-put-string! ... 68
mmap-read-position ... 67
mmap-read-position-set! ... 67
mmap-ref ... 68
mmap-set! ... 68
mmap-substring ... 68
mmap-substring-set! ... 68
mmap-write-position ... 67
mmap-write-position-set! ... 67
mmap? ... 67
module ... 7
module access file ... 16
Module body language ... 25
module class ... 12
module declaration ... 7
module export-all ... 12
module export-exports ... 12
module export-module ... 12
modules ... 7
Modules and DSSSL formal argument lists ... 24
modulo ... 35
month-aname ... 97
month-name ... 97
mp3-musictag ... 204
mpc::music ... 207
mpd ... 210
mpd-database ... 210
mpg123::musicproc ... 207
mplayer::musicproc ... 207
Multi-line comments ... 17
multimedia ... 203
multiple-value-bind ... 53
Music ... 204
music ... 206
music-close ... 208
music-closed? ... 208
music-crossfade ... 208
music-event-loop ... 209
music-meta ... 209
music-next ... 208
music-pause ... 208
music-play ... 208
music-playlist-add! ... 208
music-playlist-clear! ... 208
music-playlist-delete! ... 208
music-playlist-get ... 208
music-prev ... 208
music-random-set! ... 208
music-repeat-set! ... 208
music-reset! ... 208
music-reset-error! ... 209
music-seek ... 208
music-song ... 209
music-songpos ... 209
music-status ... 208
music-stop ... 208
music-update-status! ... 208
music-volume-get ... 208
music-volume-set! ... 208
musicproc::music ... 207
musicstatus ... 207
musictag ... 204
mutex-lock! ... 182
mutex-name ... 182
mutex-specific ... 182
mutex-specific-set! ... 182
mutex-state ... 182, 195, 196
mutex-unlock! ... 182
mutex? ... 182

N

Name mangling ... 244
nanfl? ... 36
nanoeconds->date ... 96
native-repl-printer ... 224
negative? ... 33
negativebx? ... 33
negativeelong? ... 33
negativefl? ... 33
negativefx? ... 33
negativellong? ... 33
negbx ... 35
negelong ... 34
negfl ... 34
negfx ... 34
negllong ... 34
newline ... 64
not ... 25
null-environment ... 223
null? ... 27
number->string ... 36
number? ... 32
Numbers ... 32

O

obj ... 238
obj->string ... 70
Object ... 117
object dumping ... 70
Object library ... 124
Object serialization ... 126
Object System ... 117
object->struct ... 125
object-class ... 125
object-constructor ... 125
object-display ... 125
object-equal? ... 125
object-hashnumber ... 76, 125
object-write ... 125
octet-string->bignum ... 37
odd? ... 33
oddbx? ... 33
oddelong? ... 33
oddfl? ... 33
oddfx? ... 33
oddllong? ... 33
ogg-musictag ... 204
on-line documentation ... 307
open-input-binary-file ... 70
open-input-c-string ... 58
open-input-file ... 56
open-input-ftp-file ... 58
open-input-gzip-file ... 57
open-input-gzip-port ... 57
open-input-inflate-file ... 68
open-input-procedure ... 58
open-input-string ... 58
open-input-string! ... 58
open-input-zlib-file ... 57
open-input-zlib-port ... 57
open-mmap ... 67
open-output-binary-file ... 70
open-output-file ... 59
open-output-procedure ... 59
open-output-string ... 59
open-pipes ... 66
open-semaphore ... 196
openlog ... 79
OpenPGP ... 165
operating system interface ... 76
operator ... 18
option ... 13
or ... 20
os-arch ... 78
os-charset ... 78
os-class ... 77
os-name ... 78
os-tmp ... 78
os-version ... 78
output-byte ... 71
output-char ... 71
output-obj ... 70
output-port-close-hook ... 55
output-port-close-hook-set! ... 55
output-port-flush-buffer ... 55
output-port-flush-buffer-set! ... 55
output-port-flush-hook ... 54
output-port-flush-hook-set! ... 54
output-port-name ... 54
output-port-name-set! ... 54
output-port-position ... 60
output-port-timeout-set! ... 54
output-port? ... 54
output-string ... 71
output-string-port? ... 54
Overview of Bigloo ... 3

P

pair-nil ... 238
pair-or-null? ... 27
pair? ... 27
Pairs and lists ... 27
Parallelism ... 193
parameters ... 231
password ... 66
path-separator ... 78
pattern matching ... 107
peek-byte ... 61
peek-char ... 61
pem-decode-port ... 98
pem-read-file ... 98
Performance of the JVM back-end ... 253
pgp-add-key-to-db ... 168
pgp-add-keys-to-db ... 168
pgp-db-print-keys ... 168
pgp-decrypt ... 167
pgp-encrypt ... 166
pgp-key->string ... 168
pgp-key-fingerprint ... 168
pgp-key-id ... 168
pgp-key? ... 168
pgp-make-key-db ... 168
pgp-password-encrypt ... 166
pgp-read-file ... 165
pgp-read-port ... 165
pgp-read-string ... 165
pgp-resolve-key ... 168
pgp-sign ... 167
pgp-signature-message ... 168
pgp-subkey->string ... 168
pgp-subkey? ... 168
pgp-subkeys ... 168
pgp-verify ... 167
pgp-write-file ... 165
pgp-write-port ... 165
pgp-write-string ... 165
Photography ... 203
PKCS1-v1.5-pad ... 162
PKCS1-v1.5-unpad ... 162

port->gzip-port ... 68
port->inflate-port ... 68
port->list ... 63
port->sexp-list ... 63
port->string-list ... 63
port->vcard::vcard ... 213
port->zlib-port ... 68
port? ... 54
positive? ... 33
positivebx? ... 33
positiveelong? ... 33
positivefl? ... 33
positivefx? ... 33
positivellong? ... 33
Posix Threads ... 181, 193, 196
pp ... 65
pragma::*ident* ... 244
prefix ... 79
pregexp ... 141
pregexp-match ... 142
pregexp-match-positions ... 142
pregexp-quote ... 143
pregexp-replace ... 142
pregexp-replace* ... 143
pregexp-split ... 143
pretty printing ... 307
print ... 63
printf ... 65
Private Keys ... 95
procedure call ... 18
procedure? ... 50
process ... 83
Process support ... 83
process-alive? ... 85
process-continue ... 86
process-error-port ... 85
process-exit-status ... 86
process-input-port ... 85
process-kill ... 86
process-list ... 86
process-output-port ... 85
process-pid ... 85
process-send-signal ... 86
process-stop ... 86
process-wait ... 85
process? ... 85
profiling ... 308
Program Structure ... 7
Public Key Cryptography ... 160
putenv ... 77
putprop! ... 31
pwd ... 80

Q

qualified notation ... 15
quasiquote ... 22
quit ... 224
quotation ... 22
quote ... 18
quoted-printable ... 211
quoted-printable-decode ... 211
quoted-printable-decode-port ... 211
quoted-printable-encode ... 211
quoted-printable-encode-port ... 211
quotient ... 35
quotientelong ... 35
quotientllong ... 35

R

raise ... 175
random ... 35
randombx ... 35
randomfl ... 35
rational? ... 32
read ... 60
read eval print loop ... 223
Read Eval Print Loop customized ... 223
read-byte ... 61
read-case-insensitive ... 60
read-case-sensitive ... 60
read-certificate ... 95
read-char ... 61
read-chars ... 62
read-chars! ... 62
read-csv-record ... 221
read-csv-records ... 221
read-fill-string! ... 63
read-line ... 62
read-line-newline ... 62
read-lines ... 62
read-m3u ... 204
read-of-strings ... 62
read-pem-file ... 95
read-pem-key ... 164
read-pem-key-file ... 164
read-pem-key-port ... 164
read-pem-key-string ... 164
read-private-key ... 95
read-string ... 62
read/case ... 60
read/lalrp ... 60, 139
read/rp ... 60, 129
reading path ... 16
real->string ... 36
real? ... 32
receive ... 53
records ... 113
reduce ... 29
register-class-serialization! ... 72
register-compile-srfi! ... 271

register-crc! ... 101
register-custom-serialization! ... 71
register-exit-function! ... 77
register-opaque-serialization! ... 71
register-procedure-serialization! ... 71
register-process-serialization! ... 71
register-srfi! ... 271
Regular analyser ... 129
Regular expressions ... 141
Regular parsing ... 129
regular-grammar ... 129
relative-file-name ... 81
remainder ... 35
remainderelong ... 35
remainderfl ... 35
remainderllong ... 35
remprop! ... 31
remq ... 28
remq! ... 28
rename-file ... 82
repl ... 223
reset-output-port ... 64
reverse ... 27
reverse! ... 27
revised(5) macro expansion ... 229
revision control ... 308
rfc2047-decode ... 212
rfc2047-decode-port ... 212
rfc2822-address-display-name ... 213
rfc2822-date->date ... 97
rfc2822-parse-date ... 97
RFC 2045 ... 211
RFC 2426 ... 212
RFC 2822 ... 213
RFC 3501 ... 214
rgb-hsl ... 210
rgb-hsv ... 210
rgc-context ... 135
round ... 35
roundfl ... 35
rsa-decrypt ... 161
rsa-encrypt ... 161
rsa-key-length ... 160
rsa-key=? ... 160
rsa-sign ... 161
rsa-verify ... 161
RSA-Key ... 160
RSADP ... 162
RSAEP ... 162
RSAES-OAEP-decrypt ... 163
RSAES-OAEP-encrypt ... 163
RSAES-PKCS1-v1.5-decrypt ... 163
RSAES-PKCS1-v1.5-encrypt ... 163
RSASP1 ... 162
RSASSA-PKCS1-v1.5-sign ... 163
RSASSA-PKCS1-v1.5-sign-bignum ... 163
RSASSA-PKCS1-v1.5-verify ... 163
RSASSA-PKCS1-v1.5-verify-bignum ... 163
RSASSA-PSS-sign ... 163
RSASSA-PSS-verify ... 163
RSAVP1 ... 162
run-process ... 83
run-process and char-ready? ... 61
run-process and input/output ... 61

S

safety ... 276
scheduler-broadcast! ... 193
scheduler-instant ... 192
scheduler-react! ... 191
scheduler-start! ... 192
scheduler-strict-order? ... 191
scheduler-strict-order?-set! ... 191
scheduler-terminate! ... 192
scheduler? ... 191
scheme-report-environment ... 223
searching for source code ... 308
seconds->date ... 96
seconds->string ... 96
seconds->utc-string ... 96
seed-random! ... 35
select ... 66
select (unix-like) ... 189
semaphore-post ... 196
semaphore-trywait ... 196
semaphore-value ... 196
semaphore-wait ... 196
semaphore? ... 196
send-chars ... 63
send-file ... 63
Serialization ... 70
set! ... 19
set-car! ... 27
set-cdr! ... 27
set-input-port-position! ... 60
set-output-port-position! ... 60
set-prompter! ... 224
set-read-syntax! ... 61
set-repl-error-notifier! ... 224
set-repl-printer! ... 224
setgid ... 78
setuid ... 78
sha1 ... 98
sha1sum ... 99
sha1sum-file ... 99
sha1sum-mmap ... 99
sha1sum-port ... 99
sha1sum-string ... 99
sha256 ... 98
sha256sum ... 99
sha256sum-file ... 99
sha256sum-mmap ... 99
sha256sum-port ... 99
sha256sum-string ... 99
shrink! ... 122

signal ... 77
signbitfl ... 36
sin ... 35
sinfl ... 35
sleep ... 77
Socket support ... 86
socket-accept ... 88
socket-client? ... 87
socket-close ... 89
socket-down? ... 89
socket-host-address ... 87
socket-hostname ... 87
socket-input ... 87
socket-local-address ... 87
socket-option ... 91
socket-option-set! ... 91
socket-output ... 87
socket-port-number ... 87
socket-server? ... 87
socket-shutdown ... 89
socket? ... 87
sort ... 51
soundcard::mixer ... 205
sqlite ... 199
sqlite-close ... 199
sqlite-eval ... 200
sqlite-exec ... 200
sqlite-format ... 199
sqlite-last-insert-rowid ... 201
sqlite-map ... 200
sqlite-name-of-tables ... 201
sqlite-table-name-of-columns ... 201
SQLITE ... 199
sqrt ... 36
sqrtfl ... 36
srfi-0 ... 269
srfi-0:bigloo ... 269
srfi-18 ... 174
SRFI ... 269
SRFI-0:bigloo-c ... 248
SRFI-0:bigloo-jvm ... 248
SRFI-1 ... 28, 29, 30, 51, 272
SRFI-10 ... 60
SRFI-13 ... 40, 41
SRFI-18 ... 173, 174, 193
SRFI-2 ... 19
SRFI-22 ... 272
SRFI-28 ... 64
SRFI-30 ... 17
SRFI-34 ... 173
SRFI-4 ... 49
SRFI-43 ... 48
SRFI-6 ... 58, 59
SRFI-8 ... 53
SRFI-9 ... 113
ssl-socket? ... 92
ssl-version ... 92
SSL ... 92
SSL Sockets ... 92
SSL support ... 92
Stack allocation ... 275
static ... 10
string ... 39
string escape characters ... 231
string->bignum ... 37
string->elong ... 37
string->key-hash ... 159
string->key-iterated-salted ... 159
string->key-salted ... 159
string->key-simple ... 159
string->key-zero ... 159
string->keyword ... 32
string->list ... 42
string->llong ... 37
string->mmap ... 67
string->number ... 37
string->obj ... 70
string->real ... 37
string->symbol ... 31
string->symbol-ci ... 31
string->vcard::vcard ... 213
string-append ... 42
string-capitalize ... 43
string-capitalize! ... 43
string-case ... 133
string-char-index ... 40
string-ci<=? ... 40
string-ci<? ... 40
string-ci=? ... 40
string-ci>=? ... 40
string-ci>? ... 40
string-compare3 ... 41
string-compare3-ci ... 41
string-contains ... 41
string-contains-ci ... 41
string-copy ... 42
string-cut ... 43
string-delete ... 43
string-downcase ... 42
string-downcase! ... 43
string-fill! ... 42
string-for-read ... 43
string-hash ... 76
string-hex-extern ... 44
string-hex-intern ... 44
string-hex-intern! ... 44
string-index ... 40
string-index-right ... 40
string-length ... 39
string-natural-compare3 ... 41
string-natural-compare3-ci ... 41
string-null? ... 39
string-prefix-ci? ... 44
string-prefix-length ... 44
string-prefix-length-ci ... 44
string-prefix? ... 44

string-ptr-null? ... 240
string-ref ... 39
string-replace ... 43
string-replace! ... 43
string-set! ... 39
string-shrink! ... 42
string-skip ... 40
string-skip-right ... 40
string-split ... 43
string-suffix-ci? ... 44
string-suffix-length ... 44
string-suffix-length-ci ... 44
string-suffix? ... 44
string-upcase ... 42
string-upcase! ... 43
string<=? ... 40
string<? ... 40
string=? ... 39
string>=? ... 40
string>? ... 40
string? ... 39
string2key ... 159
Strings ... 39
struct->object ... 125
struct? ... 113
structures ... 113
substring ... 42
substring-at? ... 40
substring-ci-at? ... 40
substring-ci=? ... 40
substring=? ... 40
suffix ... 80
symbol->keyword ... 32
symbol->string ... 30
symbol-append ... 31
symbol-plist ... 31
symbol? ... 30
symbols ... 30
Symmetric Block Ciphers ... 155
synchronize ... 183
Syntax ... 17
syntax-rules ... 229
syslog ... 79
syslog-facility ... 79
syslog-level ... 79
syslog-option ... 79
system ... 77
System programming ... 76
system->string ... 77

T

TAGvector ... 50
TAGvector->list ... 50
TAGvector-length ... 50
TAGvector-ref ... 50
TAGvector-set! ... 50
TAGvector? ... 50
take ... 28
tan ... 36
tanfl ... 36
tar ... 69
tar-header-checksum ... 69
tar-header-devmajor ... 69
tar-header-devminir ... 69
tar-header-gid ... 69
tar-header-gname ... 69
tar-header-linkname ... 69
tar-header-mode ... 69
tar-header-mtim ... 69
tar-header-name ... 69
tar-header-size ... 69
tar-header-type ... 69
tar-header-uid ... 69
tar-header-uname ... 69
tar-read-block ... 69
tar-read-header ... 69
tar-round-up-to-record-size ... 69
terminated-thread-exception? ... 193, 195
text ... 219
the bee root directory ... 306
The compiler environment and options ... 275
the interpreter ... 223
the lalr(1) parsing function ... 139
the pattern language ... 108
The runtime-command file ... 277
the semantics actions ... 133
The syntax of the foreign declarations ... 235
The syntax of the regular grammar ... 129
The very dangerous "pragma"
  Bigloo special forms ... 244
the-byte ... 134
the-byte-ref ... 134
the-character ... 134
the-context ... 135
the-downcase-keyword ... 134
the-downcase-symbol ... 134
the-failure ... 134
the-fixnum ... 134
the-flonum ... 134
the-keyword ... 134
the-length ... 133
the-port ... 133
the-string ... 133
the-substring ... 134
the-subsymbol ... 134
the-symbol ... 134
the-upcase-keyword ... 134
the-upcase-symbol ... 134

thread-await! ... 187
thread-await*! ... 189
thread-await-values! ... 189
thread-await-values*! ... 191
thread-cleanup ... 182
thread-cleanup-set! ... 182
thread-get-values! ... 188
thread-get-values*! ... 190
thread-join! ... 182, 187, 195
thread-name ... 181
thread-parameter ... 182
thread-parameter-set! ... 182
thread-resume! ... 187
thread-sleep! ... 182, 186, 194
thread-specific ... 181
thread-specific-set! ... 181
thread-start! ... 182, 186, 194
thread-start-joinable! ... 182, 194
thread-suspend! ... 187
thread-terminate! ... 182, 187, 195
thread-yield! ... 182, 186, 194
thread? ... 181
Threads ... 181, 185, 193, 196
Time ... 95
time ... 78
time->seconds ... 193
time? ... 193
trace ... 179
trace-bold ... 180
trace-color ... 180
trace-item ... 180
trace-margin ... 180
trace-margin-set! ... 180
trace-port ... 180
trace-port-set! ... 180
trace-string ... 180
transcript-off ... 227
transcript-on ... 227
tree-copy ... 30
truncate ... 35
truncate-file ... 82
truncatefl ... 35
try ... 53, 176
type ... 13
Typed identifier ... 17
typeof ... 173

U

ucs2->char ... 39
ucs2->integer ... 39
ucs2-alphabetic? ... 39
ucs2-ci<=? ... 39
ucs2-ci<? ... 39
ucs2-ci=? ... 39
ucs2-ci>=? ... 39
ucs2-ci>? ... 39
ucs2-downcase ... 39
ucs2-lower-case? ... 39
ucs2-numeric? ... 39
ucs2-string ... 44
ucs2-string->list ... 45
ucs2-string->utf8-string ... 45
ucs2-string-append ... 45
ucs2-string-ci<=? ... 45
ucs2-string-ci<? ... 45
ucs2-string-ci=? ... 44
ucs2-string-ci>=? ... 45
ucs2-string-ci>? ... 45
ucs2-string-copy ... 45
ucs2-string-downcase ... 45
ucs2-string-downcase! ... 45
ucs2-string-fill! ... 45
ucs2-string-length ... 44
ucs2-string-ref ... 44
ucs2-string-set! ... 44
ucs2-string-upcase ... 45
ucs2-string-upcase! ... 45
ucs2-string<=? ... 45
ucs2-string<? ... 44
ucs2-string=? ... 44
ucs2-string>=? ... 45
ucs2-string>? ... 44
ucs2-string? ... 44
ucs2-substring ... 45
ucs2-upcase ... 39
ucs2-upper-case? ... 39
ucs2-whitespace? ... 39
ucs2<=? ... 38
ucs2<? ... 38
ucs2=? ... 38
ucs2>=? ... 38
ucs2>? ... 38
ucs2? ... 38
UCS-2 characters ... 38
UCS-2 strings ... 44
unbufferized socket port ... 86, 92, 94
uncaught-exception-reason ... 193, 195
uncaught-exception? ... 193, 195
unix-path->list ... 78
unread-char! ... 58
unread-string! ... 58
unread-substring! ... 58
unregister-compile-srfi! ... 271
unregister-eval-srfi! ... 271
unregister-exit-function! ... 77
unregister-srfi! ... 271
unsigned->string ... 36
unspecified ... 53
untar ... 70
unwind-protect ... 52
uri-decode ... 103
uri-decode! ... 103
uri-decode-component ... 103
uri-decode-component! ... 103
uri-encode ... 103

uri-encode-component 103
url 102
url-decode 103
url-decode! 103
url-encode 103
url-parse 102
url-path-encode 103
url-sans-protocol-parse 102
use 9
user extensions 303
Using C bindings within the interpreter 246
utf8->8bits 46
utf8->8bits! 46
utf8->cp1252 46
utf8->cp1252! 46
utf8->iso-latin 46
utf8->iso-latin! 46
utf8->iso-latin-15 46
utf8->iso-latin-15! 46
utf8-codepoint-length 46
utf8-string->ucs2-string 45
utf8-string-encode 45
utf8-string-length 45
utf8-string-ref 46
utf8-string? 45
utf8-substring 46

V

values 53
vcard 212
VCARD 212
vector 47
vector->list 47
vector-append 48
vector-copy 48
vector-copy! 48
vector-fill! 47
vector-for-each 48
vector-length 47
vector-map 48
vector-map! 48
vector-ref 47
vector-set! 47
vector-shrink! 48
vector? 47
vectors 47
void*-null? 240
vorbis::musictag 205

W

warning 177
warning-notify 174
warning/location 174
weakptr-data 73
weakptr? 73
wide-class 117
wide-object? 125
widen! 122
widen!::*wide-class* 122
Widening and shrinking 122
with 9
with-access 119
with-access::*class* 119
with-append-to-file 55
with-error-to-file 55
with-error-to-port 56
with-error-to-procedure 56
with-error-to-string 56
with-exception-handler 174
with-handler 174
with-input-from-file 55
with-input-from-port 56
with-input-from-procedure 55
with-input-from-string 55
with-lock 183
with-output-to-file 55
with-output-to-port 56
with-output-to-procedure 55
with-output-to-string 55
with-trace 180
write 63
write-byte 64
write-char 64
write-circle 65
write-m3u 204
write-pem-key 165
write-pem-key-file 165
write-pem-key-port 165
write-pem-key-string 165

Z

zero? 33
zerobx? 33
zeroelong? 33
zerofl? 33
zerofx? 33
zerollong? 33
zip 57, 68
zlib 68

36 Library Index

(Index is nonexistent)

Bibliography

- [Bobrow et al. 88] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales and D. Moon. '`Common lisp object system specification.`' In *special issue*, number 23 in SIGPLAN Notices, September 1988.
- [BoehmWeiser88] H.J. Boehm and M. Weiser. '`Garbage collection in an unco-operative environment.`' *Software—Practice and Experience*, 18(9):807–820, Sept-ember 1988.
- [Boehm91] H.J. Boehm. '`Space efficient conservative garbage collection.`' In *Conference on Programming Language Design and Implementation*, number 28, 6 in SIGPLAN Notices, pages 197–206, 1991.
- [Caml-light] P. Weis and X. Leroy. '`Le langage CAML`'. *InterEditions, Paris, 1993.*
- [Dsssl96] ISO/IEC. '`Information technology, Processing Languages, Document Style Semantics and Specification Languages (dsssl).`' *Technical Report 10179 :1996(E), ISO, 1996.*
- [Dybvig et al. 86] K. Dybvig, D. Friedman, and C. Haynes. '`Expansion-passing style: Beyond conventional macros.`' In *Conference on Lisp and Functional Programming*, pages 143–150, 1986.
- [Gallesio95] E. Gallesio. '`STk Reference Manual.`' Technical Report RT 95-31a, I3S-CNRS/University of Nice–Sophia Antipolis, July 1995.
- [IsoC] ISO/IEC. '`9899 programming language --- C.`' *Technical Report DIS 9899, ISO, July 1990.*
- [Les75] M.E. Lesk. '`Lex --- a lexical analyzer generator.`' Computing Science Technical Report 39~39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.
- [Queinnec93] C. Queinnec. '`Designing` MEROON `v3.`' In *Workshop on Object-Oriented Programming in Lisp*, 1993.
- [QueinnecGeffroy92] C. Queinnec and J-M. Geffroy. '`Partial evaluation applied to symbolic pattern matching with intelligent backtrack.`' In M. Billaud, P. Casteran, MM. Corsini, K. Musumbu, and A. Rauzy: Editors, *Workshop on Static Analysis*, number 81-82 in bigre, pages 109–117, Bordeaux (France), September 1992.
- [R5RS] R Kelsey, W. Clinger and J. Rees: Editors. '`The Revised(5) Report on the Algorithmic Language Scheme`'.
- [Stallman95] R. Stallman. '`Using and Porting GNU CC.`' for version 2.7.2 ISBN 1-882114-66-3, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, November 1995.
- [SerranoWeis94] M. Serrano and P. Weis. '`1+1=1: an optimizing Caml compiler.`' In *ACM SIGPLAN* Workshop on ML and its Applications, pages 101–111, Orlando (Florida, USA), June 1994. ACM SIGPLAN, INRIA RR 2265.
- [Steele90] G. Steele. '`COMMON LISP (The language)`'. *Digital Press (DEC)*, Burlington MA (USA), 2nd edition, 1990.

37 Table of contents

Table of Contents

Acknowledgements **1**

1 Overview of Bigloo **3**

1.1 SRFI 3
1.2 Separate compilation 3
1.3 C interface 4
1.4 Java interface 4
1.5 Object language 4
1.6 Threads 4
1.7 SQL 4
1.8 Type annotations 4
1.9 Unicode support 5
1.10 DSSSL 5

2 Modules **7**

2.1 Program Structure 7
2.2 Module declaration 7
2.3 Module initialization 13
2.4 Qualified notation 15
2.5 Inline procedures 15
2.6 Module access file 16
2.7 Reading path 16

3 Core Language **17**

3.1 Syntax 17
3.1.1 Comments 17
3.1.2 Expressions 17
3.1.3 Definitions 22

4 DSSSL support **23**

4.1 DSSSL formal argument lists 23
4.2 Modules and DSSSL formal argument lists 24

5 Standard Library **25**

5.1 Scheme Library 25
5.1.1 Booleans 25
5.1.2 Equivalence predicates 25
5.1.3 Pairs and lists 27
5.1.4 Symbols 30
5.1.5 Keywords 31
5.1.6 Numbers 32

5.1.7 Characters ... 38
5.1.8 UCS-2 Characters ... 38
5.1.9 Strings ... 39
5.1.10 Unicode (UCS-2) Strings ... 44
5.1.11 Vectors ... 47
5.1.12 Homogeneous Vectors (SRFI-4) ... 49
5.1.13 Control features ... 50
5.2 Input and output ... 54
5.2.1 Library functions ... 54
5.2.2 mmap ... 67
5.2.3 Zip ... 68
5.2.4 Tar ... 69
5.3 Serialization ... 70
5.4 Bit manipulation ... 72
5.5 Weak Pointers ... 73
5.6 Hash Tables ... 73
5.7 System programming ... 76
5.7.1 Operating System interface ... 76
5.7.2 Files ... 79
5.7.3 Process support ... 83
5.7.4 Socket support ... 86
5.7.5 SSL ... 92
5.7.5.1 SSL Sockets ... 92
5.7.5.2 Certificates ... 95
5.7.5.3 Private Keys ... 95
5.8 Date ... 95
5.9 Digest ... 98
5.10 Cyclic Redundancy Check (CRC) ... 99
5.11 Internet ... 102
5.12 URLs ... 102
5.13 HTTP ... 103

6 Pattern Matching ... 107
6.1 Bigloo pattern matching facilities ... 107
6.2 The pattern language ... 108

7 Fast search ... 109
7.1 Knuth, Morris, and Pratt ... 109
7.2 Boyer - Moore ... 110
7.3 Boyer - Moore - Horspool ... 110

8 Structures and Records ... 113
8.1 Structures ... 113
8.2 Records (SRFI-9) ... 113

9 Object System 117
9.1 Class declaration 117
9.2 Creating and accessing objects 119
9.3 Generic functions 121
9.4 Widening and shrinking 122
9.5 Object library 124
9.5.1 Classes handling 125
9.5.2 Object handling 125
9.6 Object serialization 126
9.7 Equality 126
9.8 Introspection 126

10 Regular parsing 129
10.1 A new way of reading 129
10.2 The syntax of the regular grammar 129
10.3 The semantics actions 133
10.4 Options and user definitions 135
10.5 Examples of regular grammar 135
10.5.1 Word count 136
10.5.2 Roman numbers 136

11 Lalr(1) parsing 137
11.1 Grammar definition 137
11.2 Precedence and associativity 138
11.3 The parsing function 139
11.4 The regular grammar 139
11.5 Debugging Lalr Grammars 139
11.6 A simple example 140

12 Posix Regular Expressions 141
12.1 Regular Expressions Procedures 141
12.2 Regular Expressions Pattern Language 143
12.2.1 Basic assertions 144
12.2.2 Characters and character classes 144
12.2.3 Some frequently used character classes 145
12.2.4 POSIX character classes 145
12.2.5 Quantifiers 146
12.2.6 Numeric quantifiers 146
12.2.7 Non-greedy quantifiers 146
12.2.8 Clusters 146
12.2.9 Backreferences 147
12.2.10 Non-capturing clusters 148
12.2.11 Cloisters 148
12.2.12 Alternation 149
12.2.13 Backtracking 150
12.2.14 Disabling backtracking 150
12.2.15 Looking ahead and behind 150

12.2.16 Lookahead ... 151
12.2.17 Lookbehind ... 151
12.3 An Extended Example ... 151

13 Command Line Parsing ... 153

14 Cryptography ... 155
14.1 Symmetric Block Ciphers ... 155
14.1.1 String to Key ... 159
14.2 Public Key Cryptography ... 160
14.2.1 Rivest, Shamir, and Adleman (RSA) ... 160
14.2.1.1 RSA Keys ... 160
14.2.1.2 RSA basic operations ... 161
14.2.1.3 Examples ... 161
14.2.1.4 RSA RFC 3447 ... 162
14.2.2 Digital Signature Algorithm (DSA) ... 163
14.2.3 ElGamal ... 164
14.2.4 PEM ... 164
14.3 OpenPGP ... 165
14.3.1 Examples ... 168
14.3.1.1 Signatures ... 168
14.3.1.2 Email Usage ... 170
14.3.1.3 Encryption ... 170
14.4 Development ... 171

15 Errors, Assertions, and Traces ... 173
15.1 Errors and Warnings ... 173
15.2 Exceptions ... 174
15.3 Deprecated try form ... 176
15.4 Assertions ... 177
15.5 Tracing ... 179

16 Threads ... 181
16.1 Thread Common Functions ... 181
16.1.1 Thread API ... 181
16.1.2 Mutexes ... 182
16.1.3 Condition Variables ... 184
16.2 Threads ... 185
16.2.1 Introduction to Fair Threads ... 185
16.2.2 Fair Threads Api ... 186
16.2.2.1 Thread ... 186
16.2.2.2 Scheduler ... 191
16.2.2.3 Signal ... 192
16.2.3 SRFI-18 ... 193
16.3 Posix Threads ... 193
16.3.1 Using Posix Threads ... 194

16.3.2 Threads ... 194
16.3.3 Mutexes ... 195
16.3.4 Condition Variables ... 195
16.3.5 Semaphores ... 196
16.3.6 SRFI-18 ... 196
16.4 Mixing Thread APIs ... 196

17 Database ... 199
17.1 SQLite ... 199

18 Multimedia ... 203
18.1 Photography ... 203
18.2 Music ... 204
18.2.1 Metadata and Playlist ... 204
18.2.2 Mixer ... 205
18.2.3 Playback ... 205
18.2.4 Music Player Daemon ... 209
18.3 Color ... 210

19 Mail ... 211
19.1 RFC 2045 – MIME, Part one ... 211
19.2 RFC 2047 – MIME, Part three ... 212
19.3 RFC 2426 – MIME, Part three ... 212
19.4 RFC 2822 – Internet Message Format ... 213
19.5 Mail servers – imap and maildir ... 214
19.5.1 Mailboxes ... 214
19.5.2 IMAP (RFC 3501) ... 216
19.5.3 Maildir ... 217

20 Text ... 219
20.1 BibTeX ... 219
20.2 Character strings ... 219
20.3 Character encodings ... 220

21 CSV ... 221
21.1 Overview ... 221
21.2 API Reference ... 221

22 Eval and code interpretation ... 223
22.1 Eval compliance ... 223
22.2 Eval standard functions ... 223
22.3 Eval command line options ... 227
22.4 Eval and the foreign interface ... 227

23 Macro expansion ... 229
23.1 Expansion passing style macros ... 229
23.2 Revised(5) macro expansion ... 229

24 Parameters ... 231

25 Explicit typing ... 233

26 The C interface ... 235
26.1 The syntax of the foreign declarations ... 235
26.1.1 Automatic extern clauses generation ... 236
26.1.2 Importing an extern variable ... 236
26.1.3 Importing an extern function ... 236
26.1.4 Including an extern file ... 236
26.1.5 Exporting a Scheme variable ... 237
26.1.6 Defining an extern type ... 237
26.1.6.1 Atomic types ... 237
26.1.6.2 Struct and Union types ... 238
26.1.6.3 C pointers ... 239
26.1.6.4 C null pointers ... 240
26.1.6.5 C arrays ... 241
26.1.6.6 C functions ... 242
26.1.6.7 C enums ... 242
26.1.6.8 C opaques ... 243
26.2 The very dangerous "pragma" Bigloo special forms ... 244
26.3 Name mangling ... 244
26.4 Embedded Bigloo applications ... 245
26.5 Using C bindings within the interpreter ... 246

27 The Java interface ... 247
27.1 Compiling with the JVM back-end ... 247
27.1.1 Compiler JVM options ... 247
27.1.2 Compiling multi-modules applications ... 247
27.2 JVM back-end and SRFI-0 ... 248
27.3 Limitation of the JVM back-end ... 248
27.4 Connecting Scheme and Java code ... 249
27.4.1 Automatic Java clauses generation ... 249
27.4.2 Declaring Java classes ... 249
27.4.3 Declaring abstract Java classes ... 250
27.4.4 Extending Java classes ... 250
27.4.5 Declaring Java arrays ... 250
27.4.6 Exporting Scheme variables ... 251
27.4.7 Bigloo module initialization ... 253
27.5 Performance of the JVM back-end ... 253

28 Bigloo Libraries ... 255
28.1 Compiling and linking with a library ... 255
28.2 Library and inline functions ... 257
28.3 library and eval ... 258
28.4 library and repl ... 259
28.5 Building a library ... 259
28.6 Library and modules ... 261
28.7 Library and macros ... 262
28.8 A complete library example ... 262

29 Extending the Runtime System ... 267

30 SRFIs ... 269
30.1 SRFI 0 ... 269
30.2 SRFI 1 ... 272
30.3 SRFI 22 ... 272
30.3.1 An example of SRFI-22 script ... 273
30.3.2 Lazy compilation with SRFI-22 ... 273

31 Compiler description ... 275
31.1 C requirement ... 275
31.2 JVM requirement ... 275
31.3 Linking ... 275
31.4 The compiler environment and options ... 275
31.4.1 Efficiency ... 275
31.4.2 Stack allocation ... 275
31.4.3 Genericity of arithmetic procedures ... 276
31.4.4 Safety ... 276
31.4.5 The runtime-command file ... 277
31.4.6 The Bigloo command line ... 277

32 Cross Compilation ... 299
32.1 Introduction ... 299
32.2 Building the Bigloo library for the host-platform ... 299
32.2.1 Hostsh ... 299
32.2.2 Building ... 300
32.3 Cross Compiling Bigloo Programs ... 300
32.4 Caveats ... 300
32.5 Examples ... 301

33 User Extensions ... 303
33.1 User pass ... 303

34 Bigloo Development Environment 305
34.1 Installing the Bee 305
34.2 Entering the Bee 306
34.3 The *Bee Root Directory* 306
34.4 Building a Makefile 306
34.5 Compiling 306
34.6 Interpreting 306
34.7 Pretty Printing 307
34.8 Expansing 307
34.9 On-line Documentation 307
34.10 Searching for Source Code 308
34.11 Importing and Exporting 308
34.12 Debugging 308
34.13 Profiling 308
34.14 Revision Control 308
34.15 Literate Programming 309

35 Global Index 311

36 Library Index 329

Bibliography 331

37 Table of contents 333

Short Contents

Acknowledgements 1
1 Overview of Bigloo 3
2 Modules 7
3 Core Language 17
4 DSSSL support 23
5 Standard Library 25
6 Pattern Matching 107
7 Fast search 109
8 Structures and Records 113
9 Object System 117
10 Regular parsing 129
11 Lalr(1) parsing 137
12 Posix Regular Expressions 141
13 Command Line Parsing 153
14 Cryptography 155
15 Errors, Assertions, and Traces 173
16 Threads 181
17 Database 199
18 Multimedia 203
19 Mail 211
20 Text 219
21 CSV 221
22 Eval and code interpretation 223
23 Macro expansion 229
24 Parameters 231
25 Explicit typing 233
26 The C interface 235
27 The Java interface 247
28 Bigloo Libraries 255
29 Extending the Runtime System 267
30 SRFIs 269
31 Compiler description 275
32 Cross Compilation 299

33 User Extensions 303
34 Bigloo Development Environment 305
35 Global Index 311
36 Library Index 329
Bibliography 331
37 Table of contents 333