Bigloo User's Manual

Manuel Serrano

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Bigloo User's Manual

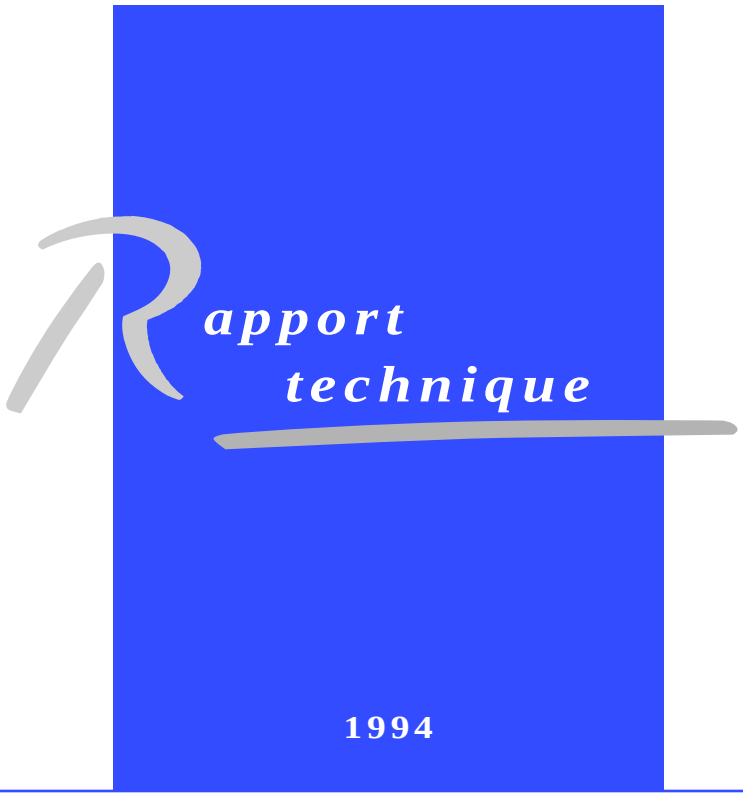
M. Serrano

N ° 169

Bigloo1.7 [07/12/94]

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel



ISSN 0249-6399

INRIA
ROCQUENCOURT

Bigloo User's Manual

M. Serrano *

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Icsla

Rapport technique n ° 169 — Bigloo1.7 [07/12/94] — 26 pages

Abstract: Bigloo is a Scheme compiler developped at Inria-Rocquencourt in the Icsla team. This is the continuation of Nitsan Séniak's work on the compilation of functional languages [Sén91], Christian Queinnec's modules for Scheme [QP91] and Jean-Marie Geffroy's pattern-matching compiler [QG92].

Key-words: Scheme, compilation, functional languages

(Résumé : tsvp)

*Manuel.Serrano@inria.fr

Manuel d'utilisation de Bigloo

Résumé : Bigloo est un compilateur Scheme developpe à l'Inria-Rocquencourt dans l'équipe Icsla. Ce travail s'inscrit dans le prolongement de ceux de Nitsan Séniak sur la compilation des langages fonctionnels, de Christian Queinnec sur les langages de modules pour Scheme [QP91] et de Jean-Marie Geffroy sur le filtrage [QG92].

Mots-clé : Scheme, compilation, langages fonctionnels

Introduction

Bigloo is a Scheme compiler developped at Inria-Rocquencourt in the Icsla team. This is the continuation of Nitsan Séniak's work on the compilation of functional languages [Sén91], Christian Queinnec's modules for Scheme [QP91] and Jean-Marie Geffroy's pattern-matching compiler [QG92].

I would like to address special thanks to Joel F. Bartlett for his Scheme-to-C compiler [Bar89]. The first version of Bigloo was compiled with it and looking at the Scheme-to-C's code was of great help. Many library functions are almost copied from it. I also thank Hans J. Boehm for his Garbage Collector [Boe91]; the current runtime uses it.

This release of Bigloo might still contain bugs. If you notice some of them, please send a mail at the following address:

`Manuel.Serrano@inria.fr`

Contents

1 Overview

Bigloo is an implementation of the Scheme language. It is both a compiler and an interpreter. Bigloo does not entirely conform to Scheme as defined in the IEEE standard for the Scheme Programming Language [11791]. The two main reasons are:

- Bigloo is a Scheme to C compiler.[1] The generated C code uses the C stack, so some programs can't be

[1] A Scheme file is compiled into a C file, which is then compiled by a C compiler.

properly tail recursive. Nevertheless all simple tail recursions are compiled without stack consumption.

- Bigloo is a module compiler. It separately compiles modules into '.o files' that must be linked together to produce stand alone executable programs.

However, we designed Bigloo to be as close as possible to the IEEE standard for the Scheme Programming Language: this documentation is therefore a mere comparison to it.

2 Bigloo vs Scheme

Here is the exhaustive list of points where Bigloo does not conform to the IEEE standard for the Scheme Programming Language.

2.1 Syntactic keywords

Bigloo has more syntactic keywords than Scheme. The syntactic keywords are:

`=>`	`do`	`or`
`and`	`else`	`quasiquote`
`begin`	`if`	`quote`
`case`	`lambda`	`set!`
`cond`	`let`	`unquote`
`define`	`let*`	`unquote-splicing`
`delay`	`letrec`	`module`
`labels`	`try`	`define-struct`
`bind-exit`	`define-inline`	`regular-grammar`
`regular-search`	`define-macro`	`define-expander`
`match-case`	`match-lambda`	`pragma`
`assert`		

All the syntactic keywords, that are located from page-14 to page-19 of the IEEE standard for the Scheme Programming Language, have the same definition in Bigloo.
The new syntactic keywords will be explained later.

2.2 Program structure

A Bigloo program is composed of several modules. The definition of a module is given in Section 3. A module is a Scheme program as defined in the IEEE standard for the Scheme Programming Language page-18.

2.3 Standard procedure

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

2.3.1 Booleans

The standard boolean objects are `#t` and `#f`.

Note: the empty list is true.
`(not` *obj*`)` procedure

2.3.2 Equivalence predicates

`(eqv?` *obj1 obj2*`)` procedure
`(eq?` *obj1 obj2*`)` procedure
`eqv?` and `eq?` are equivalent in Bigloo.

`(equal?` *obj1 obj2*`)` procedure

2.3.3 Pairs and lists

The form `()` is *illegal*.
`(pair?` *obj*`)` procedure

`(car` *pair*`)` procedure
`(cdr` *pair*`)` procedure
`(set-car!` *pair obj*`)` procedure
`(set-cdr!` *pair obj*`)` procedure

`(caar` *pair*`)` procedure
`(cadr` *pair*`)` procedure
⋮ ⋮
`(cdddar` *pair*`)` procedure
`(cddddr` *pair*`)` procedure

`(null?` *obj*`)` procedure
`(list?` *obj*`)` procedure
`(list` *obj* ...`)` procedure
`(length` *list*`)` procedure
`(append` *list* ...`)` procedure

`(reverse` *list*`)` procedure
`(reverse!` *list*`)` bigloo procedure
A destructive reverse.

`(list-ref` *list k*`)` procedure
`(list-tail` *list k*`)` bigloo procedure
Returns the sublist of *list* obtained by omitting the first *k* elements.

`(last-pair` *list*`)` procedure
Returns the last pair in the nonempty, possibly improper, *list*.

`(memq` *obj list*`)` procedure
`(memv` *obj list*`)` procedure
`(member` *obj list*`)` procedure
`(assq` *obj alist*`)` procedure

(**assv** *obj alist*) procedure
(**assoc** *obj alist*) 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.

(**remove** *obj list*) bigloo procedure
Returns a new **list** which is a copy of *list* with all items **equal?** to *obj* removed from it.

(**remove!** *obj list*) bigloo procedure
Same as **remove** 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.

2.3.4 Symbols

Symbols are case insensitive. So
`(eq? 'toto 'TOTO)` ↦ **#t**
(**symbol?** *obj*) procedure
(**symbol->string** *symbol*) procedure
(**string->symbol** *string*) procedure
(**symbol-append** *symbol* ...) bigloo procedure
Returns a symbol whose name is the concatenation of all the *symbol*'s names.

(**gensym**) bigloo procedure
(**gensym** *string*) bigloo procedure
Returns a new fresh symbol. If *string* is provided it is used as prefix for the new symbol.

(**symbol-plist** *symbol*) bigloo procedure
Returns the property-list associated with *symbol*.

(**getprop** *symbol key*) bigloo procedure
Returns the value that has the key **eq?** to *key* from the *symbol*'s property list. If there is no value associated with *key* then **#f** is returned.

(**putprop!** *symbol key val*) bigloo procedure
Stores *val* using *key* on *symbol*'s property list.

(**remprop!** *symbol key*) bigloo procedure
Removes the value associated with *key* in the *symbol*'s property list. The result is unspecified.

2.3.5 Numbers

Bigloo has only two kinds of numbers: fixnum and flonum. Operations on complexes and rationals are not implemented but for compatibility purposes, the functions **complex?** and **rational?** exist. The binary radix is not implemented so the only accepted prefixes are **#o**, **#d** and **#x**. For each generic arithmetic procedure, Bigloo provides two specialized procedures, one for fixnums and one for flonums. The name of these two specialized procedures is the name of the original one suffixed by **fx** or **fl**. A fixnum has the size of a C's **integer** minus 2 bits. A flonum has the size of a C's **double**.

(**number?** *obj*) procedure
(**real?** *obj*) procedure
(**integer?** *obj*) procedure
(**complex?** *x*) procedure
(**rational?** *x*) procedure
These two procedures always return **#t**.
(**exact?** *z*) procedure
(**inexact?** *z*) procedure

(**zero?** *z*) procedure
(**positive?** *z*) procedure
(**negative?** *z*) procedure
(**odd?** *n*) procedure
(**even?** *n*) procedure

(**max** x_1 x_2...) procedure
(**min** x_1 x_2...) procedure

(**=** z_1 z_2...) procedure
(**=fx** n_1 n_2) bigloo procedure
(**=fl** x_1 x_2) bigloo procedure
(**<** z_1 z_2...) procedure
(**<fx** n_1 n_2) bigloo procedure
(**<fl** x_1 x_2) bigloo procedure
(**>** z_1 z_2...) procedure
(**>fx** n_1 n_2) bigloo procedure
(**>fl** x_1 x_2) bigloo procedure
(**<=** z_1 z_2...) procedure
(**<=fx** n_1 n_2) bigloo procedure
(**<=fl** x_1 x_2) bigloo procedure
(**>=** z_1 z_2...) procedure
(**>=fx** n_1 n_2) bigloo procedure
(**>=fl** x_1 x_2) bigloo procedure

(**+** z_1...) procedure
(**+fx** n_1 n_2) bigloo procedure
(**+fl** x_1 x_2) bigloo procedure
(***** z_1...) procedure
(***fx** n_1 n_2) bigloo procedure
(***fl** x_1 x_2) bigloo procedure
(**-** z) procedure

`(- `z_1 $z_2 \ldots$`)` procedure
`(-fx `n_1 n_2`)` bigloo procedure
`(-fl `x_1 x_2`)` bigloo procedure
`(negfx `n`)` bigloo procedure
`(negfl `x`)` bigloo procedure
These two functions implement the unary function `-`.
`(/ `z_1 z_2`)` procedure
`(/ `z`)` procedure
`(/fx `n_1 n_2`)` bigloo procedure
`(/fl `x_1 x_2`)` bigloo procedure

`(abs `x`)` procedure
`(quotient `n_1 n_2`)` procedure
`(remainder `n_1 n_2`)` procedure
`(modulo `n_1 n_2`)` procedure
`(gcd `$n_1 \ldots$`)` procedure
`(lcm `$n_1 \ldots$`)` procedure
`(floor `x`)` procedure
`(ceiling `x`)` procedure
`(truncate `x`)` procedure
`(round `x`)` procedure

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

`(exact->inexact `z`)` procedure
`(inexact->exact `z`)` procedure
`(number->string `x`)` procedure
`(integer->string `n`)` bigloo procedure
`(integer->string `n *radix*`)` bigloo procedure
`(real->string `z`)` bigloo procedure
`(string->number `*string*`)` procedure
`(string->integer `*string*`)` bigloo procedure
`(string->integer `*string radix*`)` bigloo procedure
`(string->real `*string*`)` bigloo procedure
`(fixnum->flonum `n`)` bigloo procedure
`(flonum->fixnum `x`)` bigloo procedure
These two last procedures implement the natural translation from fixnum to flonum.

2.3.6 Characters

Bigloo knows one more named character `#\tab`. A new alternate syntax exists to read characters:

`#a<ascii-code>` where `<ascii-code>` is the decimal ascii number containing exactly three digits of the character to be read.

`(char? `*obj*`)` procedure

`(char=? `*char*$_1$ *char*$_2$`)` procedure
`(char<? `*char*$_1$ *char*$_2$`)` procedure
`(char>? `*char*$_1$ *char*$_2$`)` procedure
`(char<=? `*char*$_1$ *char*$_2$`)` procedure
`(char>=? `*char*$_1$ *char*$_2$`)` procedure
`(char-ci=? `*char*$_1$ *char*$_2$`)` procedure
`(char-ci<? `*char*$_1$ *char*$_2$`)` procedure
`(char-ci>? `*char*$_1$ *char*$_2$`)` procedure
`(char-ci<=? `*char*$_1$ *char*$_2$`)` procedure
`(char-ci>=? `*char*$_1$ *char*$_2$`)` procedure

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

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

`(char-upcase `*char*`)` procedure
`(char-downcase `*char*`)` procedure

2.3.7 Strings

There are two different syntaxes for strings in Bigloo: strings may be specified as traditional or foreign strings. The traditional syntax for strings conforms to the IEEE standard for the Scheme Programming Language. With the foreign syntax, the C escaping sequences are interpreted as specified by ISO-C. Only the reader distinguishes between these two appearance of strings, i.e., there is only one type of string at evaluation-time. The regular expression describing the syntax for foreign string is: `#"([^"]|")*"`.
For example, after reading the expression `"1\n23\b4\"5"`, the following string is built:

1	\	n	2	3	\	b	4	"	5

Printing this string will produce:

```
1\n23\b4\"5
```

The new foreign syntax allows C escaping sequences to be recognized as in: `#"1\n23\b4\"5"` This expression builds the following string:

1	#\Newline	2	3	#\bs	4	"	5

Printing this string will then produce:

```
1
23"5
```

The libraries functions on string processing are:
`(string? `*obj*`)` procedure

(make-string *k*) procedure
(make-string *k char*) procedure
(string *k* ...) procedure

(string-length *string*) procedure
(string-ref *string k*) procedure
(string-set! *string k char*) procedure

(string=? $string_1$ $string_2$) procedure
(string-ci=? $string_1$ $string_2$) procedure
(string<? $string_1$ $string_2$) procedure
(string>? $string_1$ $string_2$) procedure
(string? $string_1$ $string_2$) procedure
(string<=? $string_1$ $string_2$) procedure
(string>=? $string_1$ $string_2$) procedure
(string-ci<? $string_1$ $string_2$) procedure
(string-ci>? $string_1$ $string_2$) procedure
(string-ci<=? $string_1$ $string_2$) procedure
(string-ci>=? $string_1$ $string_2$) procedure

(substring *string start end*) procedure
(string-append *string* ...) procedure
(string->list *string*) bigloo procedure
(list->string *chars*) bigloo procedure
(string-copy *string*) 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
Builds a new allocated string in lower case letter.
(string-upcase *string*) bigloo procedure
Builds a new allocated string in upper case letter.
(string-downcase! *string*) bigloo procedure
Physically downcases the *string* argument.
(string-upcase! *string*) bigloo procedure
Physically upcases the *string* argument.

(string-for-read *string*) bigloo procedure
Returns a copy of *string* where all the special characters were replaced by escape sequences.

(blit-string! *s1 o1 s2 o2 long*) bigloo procedure
Fill string *s1* starting at position *o1* with *long* characters taken out of string *s2* from position *o2*.

2.3.8 Vectors

Vectors are not autoquoted objects.
(vector? *obj*) procedure

(make-vector *k*) procedure
(make-vector *k fill*) procedure
(vector *k* ...) procedure

(vector-length *vector*) procedure
(vector-ref *vector k*) procedure
(vector-set! *vector k obj*) procedure

(vector->list *vector*) procedure
(list->vector *obj*) procedure
(vector-fill! *vector fill*) procedure

2.3.9 Control features

(procedure? *obj*) procedure

(apply *proc args*) procedure
(apply *proc* obj_1 ... *args*) procedure

(map *proc* $list_1$ $list_2$ ...) procedure
(for-each *proc* $list_1$ $list_2$ ...) procedure

(force *promise*) procedure

(call/cc *proc*) bigloo procedure
This function is the same as the call-with-current-continuation function of the IEEE standard for the Scheme Programming Language. But, to use it, it is necessary to compile the module with the -call/cc option (see Section 13).

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.

(labels *bindings body*) bigloo syntax
The syntax is similar to the Common Lisp one [Ste90], where created bindings are immutable.

```
(labels ((var1 (arg1 ...) body1) ...)
        body)
```

(bind-exit (*escape*) *body*) bigloo syntax
This 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.

(unspecified) bigloo procedure
Returns an 'unspecified' object with no specific property.

2.3.10 Input and output

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

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

`(file-exists?` *string*`)` bigloo procedure
This procedure returns `#t` if the file *string* exists. Otherwise it returns `#f`.

`(delete-file` *string*`)` bigloo procedure
Deletes the file named `string`. The result of this procedure is unspecified.

`(rename-file` $string_1$ $string_2$`)` bigloo procedure
Renames the file $string_1$ as $string_2$. If the renaming succeeds, the result is `#t`, otherwise it is `#f`.

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

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

`(open-input-file` *fname*`)` procedure
`(open-input-string` *string*`)` bigloo procedure
Returns an `input-port` able to deliver characters from *string*.

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

`(binary-port?` *obj*`)` procedure
`(open-output-binary-file` *fname*`)` bigloo procedure
`(append-output-binary-file` *fname*`)` bigloo procedure
`(open-input-binary-file` *fname*`)` bigloo procedure
`(close-binary-file` *fname*`)` bigloo procedure
`(input-obj` *port*`)` bigloo procedure
`(output-obj` *port*`)` bigloo procedure
Bigloo allows to dump into files and restore from them Scheme objects. These operations are performed by the previous functions. The dump and the restore use the two functions `obj->string` and `string->obj`.

`(open-output-string)` bigloo procedure
This function returns an *output string port*. This object has almost the same purpose as the `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 invokation of `flush` or `close-output-port` on an *output string port* returns a new string which contains all the characters accumulated in the port.

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

`(read)` procedure
`(read` *port*`)` procedure
`(read/rp` *grammar port*`)` bigloo procedure
This function is fully explained in the Section 4.

`(read-char)` procedure
`(read-char` *port*`)` procedure
`(peek-char)` procedure
`(peek-char` *port*`)` procedure
`(eof-object?` *obj*`)` procedure
`(char-ready?)` procedure
`(char-ready?` *port*`)` procedure
This function always returns `#t`.

`(read-string` *port n*`)` bigloo procedure
Returns a new allocated string composed of the first *n* characters read from the *port*. If less than *n* characters can be read, the string will have the size of the number of characters actually read.

`(read-line` *port*`)` bigloo procedure
Reads characters from *port* until a `#\Newline` or an `end of file` condition is encountered. `read-line` returns a new allocated string composed of the characters read.

`(read-of-strings` *input-port*`)` bigloo procedure
Reads sequence of non space characters on *input-port* and makes strings of them.

`(write` *obj*`)` procedure
`(write` *obj port*`)` procedure
`(display` *obj*`)` procedure
`(display` *obj port*`)` procedure
`(print` *obj* ...`)` bigloo procedure
This procedure allows to display several objects. When all these objects are printed, `print` adds a newline.

`(display*` *obj* ...`)` bigloo procedure
This function is similar to `print` but does not add a newline.

`(fprint` *port obj* ...`)` bigloo procedure
This function is the same as `print` except that a port is provided.

`(newline)` procedure
`(newline` *port*`)` procedure
`(write-char` *char*`)` procedure
`(write-char` *char port*`)` procedure
`(flush-output-port` *port*`)` bigloo procedure
This procedure flushes an output-port.

`(set-write-length!` *len*`)` bigloo procedure
Sets to *len* the maximum number of atom that can be printed by `write` and `display`. This facility is useful to prevent the printer to fall into infinite loop when printing

circular structures.

`(get-write-length)` bigloo procedure
Gets the current length of the printer.

`(pp` *obj* `.` *port*`)` bigloo procedure
Pretty print *obj* on *port*.

`*pp-case*` bigloo variable
Sets the variable to `respect` or `upper` to change the case of the pretty-print.

`*pp-width*` bigloo variable
The width of the pretty-print.

`(string->obj` *object*`)` bigloo procedure
This function converts a *string* which has been produced by `obj->string` into a Bigloo object.

`(obj->string` *string*`)` bigloo procedure
This function converts *any* Bigloo *object* which does not contain procedure in a string.
The implementation of the two last functions ensures that for every Bigloo object `obj` (containing no procedure), the expression:

```
(equal? obj (obj->string (string->obj obj)))
```

always evaluates to `#t`.

2.3.11 Bit manipulation

These procedures allow to manipulate fixnums as bit-fields.

`(bit-or` z_1 z_2`)` bigloo procedure
`(bit-xor` z_1 z_2`)` bigloo procedure
`(bit-and` z_1 z_2`)` bigloo procedure
`(bit-not` *z*`)` bigloo procedure
`(bit-lsh` z_1 z_2`)` bigloo procedure
`(bit-rsh` z_1 z_2`)` bigloo procedure

2.3.12 System interface

`*load-path*` bigloo variable
A list of search paths for the `load` functions.

`(load` *filename*`)` bigloo procedure
`(loadq` *filename*`)` bigloo procedure
Filename should be a string naming an existing file containing Bigloo source code. This file is searched in the current directory and in all the directories contained in the variable `*load-path*`. The `load` procedure reads expressions and definitions from the file and evaluates them sequentially. If the file loaded is a module (i.e. if the file loaded begins with a regular module clause), load behaves as module initialization. Otherwise, this function returns the result of the last evaluation.

`(loada` *filename*`)` bigloo procedure
Loads an "access file" to allow the interpreter to find where are the modules imported by a loaded module.

`(exit` *int*`)` bigloo procedure
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` *string*`)` bigloo procedure
Returns the integer result of the system command *string*.

`(getenv` *string*`)` bigloo procedure
Returns the string value of the Unix shell's *string* variable. If no such variable is bound, `getenv` returns `#f`.

`(date)` bigloo procedure
Returns the current date in a `string`.

`(command-line)` bigloo procedure
Returns a list containing strings which are the Unix process arguments.

2.3.13 Unix file name

Four procedures exist to manipulate Unix filenames.

`(basename` *string*`)` bigloo procedure
Returns a copy of *string* where prefix ending in `/` is deleted.

`(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")` $\mapsto$ `"foo"` .

`(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")` $\mapsto$ `"scm"` .

`(dirname` *string*`)` bigloo procedure
Returns a new string which is the directory component of *string*.

2.3.14 Hash tables

Bigloo offers hash tables. Here are described functions which define and use them.

`(make-hash-table` *ms nb gkey eq*`)` bigloo procedure
`(make-hash-table` *ms nb gkey eq is*`)` bigloo procedure
Defines an hash table of maximum length *ms*. If *is* is provided, it sets the initial table size. The formal *nb* is a function which if applied to a `key`, has to return an integer bound in interval $[0 \ldots ms]$. The formal *gkey* is a function which if applied to an object, returns its key. For example,

the identity is a good candidate for integers, strings, symbols, etc. The last formal, *eq* is a equivalence predicate which is used when searching an entry in the table.

`(hash-table?` *obj*`)` bigloo procedure
Returns `#t` if *obj* is an hash table.

`(hash-table-nb-entry` *table*`)` bigloo procedure
Returns the number of entries contained in *table*.

`(hash-table-vector` *table*`)` bigloo procedure
Returns the vector associated to *table*.

`(get-hash` *key table*`)` bigloo procedure
Returns the entry which key is *key* in *table*.

`(put-hash!` *obj table*`)` bigloo procedure
Puts *obj* in *table*. This function returns the object bound in the table. If an object with the same key was already bound, this function returns it.

`(rem-obj-hash!` *obj table*`)` bigloo procedure
Removes *obj* in *table* and returns `#t` if such object was bound in table.

`(rem-key-hash!` *key table*`)` bigloo procedure
Removes an object associated to *key* in *table* and returns `#t` if such object was bound in table.

`(for-each-hash` *fun table*`)` bigloo procedure
Applies *fun* to all the elements of *table*.

Some functions computes hash numbers from Scheme objects:

`(string->0..255` *string*`)` bigloo procedure
`(string->0..2^x-1` *string power*`)` bigloo procedure
The formal *power* has to be a 2 power between 1 and 16.

`(int->0..255` *integer*`)` bigloo procedure
`(int->0..2^x-1` *integer power*`)` bigloo procedure
`(obj->0..255` *obj*`)` bigloo procedure
`(obj->0..2^x-1` *obj power*`)` bigloo procedure

3 The module language

Bigloo is a module compiler, it compiles modules. A module contains a module declaration that defines its name, importation and exportation directives and several Bigloo expressions.

3.1 Module declaration

The module declaration form is
`(module` *name* $clause_1$ $clause_2$ ...`)` bigloo syntax
This form defines a module. It must be the first in the file. The argument *name* is a symbol, the name of the module. Each $clause_i$ can be:

`(main` *function-name*`)`
This clause defines the entry point for a stand alone application. This procedure has to be of arity one.

Note: (see also Section 13.)

`(include` $file\text{-}name_1$ $file\text{-}name_2$ ...`)`
This is a list of *file-name* which will be included in the source file. Included files have a special syntax, they are not modules. An included file contains Bigloo expressions, importation clauses and include clauses. The importation and include clauses must be written in a list where the first element is the keyword `directives`. Here is an example of an include file:

```
;; the directives
(directives (include "bar.sch")
            (import  hux))
;; expressions
(define (gee x) (print x))
```

`(load` $load_1$ $load_2$ ...`)`
A $load_i$ is a list of the form:

<load> ⟶ <lclause>*
<lclause> ⟶ (<module-name> <file>)
| <module-name>

This clause forces Bigloo to load the module specified in the `lclause` in the environment used by the macroexpansion mechanism. That means that the user's macros can use all the binding of all the loaded modules.

`(with` $module_1$ $module_2$ ...`)`
This is a list of modules which have to be loaded at runtime when the module *name* is used and which have to be linked to produce the stand alone application.

`(import` $import_1$ $import_2$ ...`)`
An $import_i$ is a list of the form:

<import> ⟶ <iclause>*
<iclause> ⟶ (<variable> <module-name> <file>)
| (<variable> <module-name>)
| <module-name>
| (<module-name> <file>)
<module-name> ⟶ <variable>
<file> ⟶ <string>
<variable> ⟶ <any <identifier> that
isn't also a <syntactic keyword>>

The first form imports the variable named <variable> wich is defined in the module <module-name>, located in the file <file>. 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>. The fifth form allows the use of variables which have to be defined in the interpreter.

`(top-level` *variable*$_1$ *variable*$_2$ ...`)`
Thos form allows the use of variables which have to be defined in the interpreter.

`(use` *import*$_1$ *import*$_2$ ...`)`
`use` has the same meaning as `import` except that modules which are only used are not initialized.

`(force` *module*$_1$ *module*$_2$ ...`)`
This clause forces the initialization order of modules. The forced modules are initialized in the order they appear in this clause but *after* all non-forced modules. All the *forced* modules are initialized (even if they are just *used*).

`(export` *export*$_1$ *export*$_2$ ...`)`
An *export*$_i$ is a list of the form:

<export> ⟶ <eclause>*
<eclause> ⟶ (inline <name> <formals>)
| (<name> <formals>)
| <name>
<name> ⟶ <variable>
<formals> ⟶ <variable>*
| <variable>* . <variable>

The first form exports the <name> inline-procedure.[2] The second form lets the function <name> to be exported and the third lets the variable named <name> to be exported.

`(static` *static*$_1$ *static*$_2$ ...`)`
A *static*$_i$ has exactly the same syntax as an export clause. But the declared bindings are local to the module.

`(foreign` *foreign*$_1$ *foreign*$_2$ ...`)`
Foreign clauses will be explained later, in Section 9.

As it can be seen, it is only necessary to give the prototype of a variable at export-time. Let's see an example of a module named `foo` which entirely imports module `bar` but only the function `fhux` of the module `hux`. The module `foo` exports `ffoo1`, by default `ffoo2` is static.

```
(module foo
        (import bar
                (fhux hux))
        (export (ffoo1  x y z)))

(define (ffoo1 x y z)
   x)

(define (ffoo2 x)
   x)
```

[2] The definition of an inline-procedure will be explained later.

3.2 Inline procedures

Bigloo has a class of procedures which are called inline. An inline procedure differs from a normal one only for the application. An inline procedure is a first order procedure which can be manipulated as any other procedure but when Bigloo sees an application of an inline procedure, 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
This form is similar to the `define` one.
Inline procedures are exportable. This means that the compiler scans the imported files to find all the inline procedures body. 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))
```

The open-coding of the importer module will lead to the following equivalent module:

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

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

Not every procedure can be an inline. Some restrictions are applied to them:

- Inline can't use global functions which are not exported.

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

3.3 Module access file

Unlike other languages (C for example), a module is not defined by a file. This means that the module name is not necessarily the name of the file where the text of the module is written.

Since modules are defined independently of files, it is necessary to make a link between them. There are two ways to do that. Choosing an import clause where the file-name is specified or creating a "module access file". This file must contain only one *list* of the form:
{*(module-name "file-name")*}*

3.4 Alias solving

When Bigloo sees several declarations of the same function (for example a module defines a function `foo` and imports a module which exports another function `foo`) it uses the last declared, knowing that all the imported variables are bound before the local functions.

The library functions are considered to be defined before all others. This means that it is possible to redefine them but if they are, the re-definition is, of course, local to the module.

3.5 The top level forms

Top level forms are put together in a single function, in the order they are read. Every initialization function of the *with* module is invoked at initialization-time. However, it's hazardous to use the value of some variables which are initialized in other modules because the order of evaluation is unspecified in Bigloo.

4 Regular parsing

Programming languages have poor reading libraries. The lexical informations that it is allowed to specify are 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. To circumvent this problem, there is a set of programs: Lex[Les75], for example, is one of them. We choose to incorporate in Bigloo a set of new functions to allow this task.

4.1 A new way of reading

There is only one way to read a text, the regular reading, which is done by the new form:

`(read/rp` *regular-parser port*`)` bigloo procedure

The first argument is a regular parser and the second a Scheme port. The way of reading is almost the same as the Lex's one. The reader tries to match the longest input, taken on the stream pointed by *port* with one or several regular expressions contained by *regular-parser*. 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))
```

4.2 The syntax of the regular parser

There are two ways of building regular parser, `regular-grammar` and `regular-search`.

A grammar is defined by the form:

`(regular-grammar` <*bdg*> <*rule*> ...`)` bigloo syntax

The *bdg* and *rules* are defined by the following grammar:

```
<bdg> ⟶ ( <variable> <reg-exp> ) *
<rule> ⟶ ( <context> <reg-exp> <sequence> )
       | ( <reg-exp> <sequence> )
       | (else <sequence> )
<context> ⟶ <symbol>
<reg-exp> ⟶ (... <uinteger R> (uncase <string> ))
       | (... <uinteger R> <string> )
       | <string>
       | <char>
       | all
       | <variable>
       | (uncase <string> )
       | (! <reg-exp> )
       | (? <reg-exp> )
       | (* <uinteger R> <reg-exp> )
       | (* <reg-exp> )
       | (+ <uinteger R> <reg-exp> )
       | (+ <reg-exp> )
       | (<-> <interval> )
       | (>-< <interval> )
       | (>-< <char> )
       | (in <char>+ )
       | (out <char>+ )
       | (bol <reg-exp> )
       | (eol <reg-exp> )
       | (eof <reg-exp> )
       | ( <reg-exp> )
<interval> ⟶ ( <char> <char> )
       | ( <char> <char> ) <interval>
```

In this grammar there exist operators (which appear in the definition of the regular expression), they allow the

construction of the expression. There are three kind of operators. The primitives:

```
*    the Kleene's operator
?    0 or 1 occurrence operator
!    the union operator
     the concatenation operator (implicit)
```

The macros which can be deduced from the primitives are:

(>-< $char_{min}$ $char_{max}$)	≡ (! $char_{min}$ ... $char_{max}$)
(<-> $char_{min}$ $char_{max}$)	≡ (out $char_{min}$ ... $char_{max}$)
(<-> char)	≡ (out char)
(uncase string)	≡ the union of all the letters in upper case or lower case of the string
(... integer string)	≡ the first one, two, ... integer letters of string
(all)	≡ (out #\newline)
(+ e)	≡ (e (* e))
(+ num e)	≡ (e ... e)
(* num e)	≡ (! (? e) (e e) ...)
(in $char_1$ $char_2$ ...)	≡ (! $char_1$ $char_2$ ...)
(out $char_1$ $char_2$ ...)	≡ (! all the characters excepted $char_1$ $char_2$ ...)

And the exceptions which are not regular constructors but rather context constructors:

context	A context is a symbol. A rule can be used to match text only if the reader has been put on this context
eol	(eol reg) Use reg only on "end of line"
bol	(bol reg) Use reg only on "beginning of line"
eof	(eof reg) Use reg only on "end of file"

A searcher is defined by the form:

(regular-search *reg-exp*) bigloo syntax

Defines a regular-parser which looks for the *reg-exp* on an input-port. If the expression is found, the parser returns the longest string which belongs to the rational language defined by *reg-exp*. Otherwise, it returns #f.

4.3 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:

(context *new-context*)	put the reader in the context *new-context*.
(context)	remove all contexts.
(the-length)	Get the length of the biggest matching string.
(the-string)	get a copy of the last matching string.
(unget-char)	unread the last char.
(ignore)	ignore the parsing, keep reading. Its 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.
(the-small-string)	the matching string where the first and the last char are removed.
(the-symbol)	the conversion of the last matching string to symbol.
(the-failing-char)	returns the first char that the grammar can't match.

4.4 An example of grammar

The reader who wants to find a real example should read the code of the Bigloo's reader. But here is a small example of 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))))
```

5 Pattern matching

Pattern matching is a key feature of most modern functional programing languages. It allows to write a clean and secure code. Internally, the "pattern-matching forms" are translated (compiled) into cascades of "elementary tests". A key point is that this translation must provide a code as efficient as possible, avoiding redundant tests. Bigloo's "pattern matching compiler" provides such a property. The technique used is described in details in [QG92], and the code can be considered as optimal[3], due to the way this "pattern compiler" was obtained.
The "pattern language" allows to express a wide variety of patterns, including:

- Non-linear patterns: pattern variables can appear more than once, allowing to compare subparts of the datum (through eq?)

[3] In the cases of pattern matching in lists and vectors, not in structures for the moment.

- Recursive patterns on lists: you can for example check 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 or structures.

5.1 Bigloo's pattern matching facilities

Only two special forms are added in Bigloo: `match-case` and `match-lambda`, which also exist for example, in Andrew Wright and Bruce Duba's [WD93] pattern matching package:

`(match-case` <key> <clause$_1$>...) syntax

Syntax: <key> may be any expression. Each <clause> shall have the form
(<pat> <expression$_1$> <expression$_2$> ...)

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

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

A dual syntax is also available:

`(match-lambda` <clause$_1$> <clause$_2$> ...) 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
```

5.2 The pattern language

The syntax is presented in figure 1. It is described in the same way (and nearly in the same words) as in [WD93].

Remark `and`, `or`, `not`, `check` and `kwote` must be quoted in order to be treated as litterals. This is the only reason which justifies the presence of the `kwote` pattern, since by convention any atom which is not a keyword is quoted.

Explanations through examples

- `?-` matches any s-expr
- `a` matches the atom `'a`.
- `?a` matches any expression, and binds the variable a to this expression.
- `(? integer?)` matches any integer
- `(a (a b))` matches the only list `'(a (a b))`.
- `???-` can only appear at the end of a list, and always succeeds. For instance, `(a ???-)` is equivalent to `(a . ?-)`.
- 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, eventually 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 *eq* to its `cadr`
- `#(?- ?- ???-)` matches any vector of length greater (or equal to) than 2.
- `#{foo (z (?- . ?-)) (x (? integer?)}` matches any structure `foo` whose fields `z` and `x` are respectively a pair and an integer. You can provide only the fields you want to test. The order is not relevant.
- Similarly, `#{(z (?- . ?-)) (x (? integer?)}` matches any structure whose fields `z` and `x` are respectively a pair and an integer. The matched structure is the latest defined structure which has fields named `x` and `z`.

Remark `??-` and `...` patterns can not appear inside a vector, where you may simply 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`.

<pattern> ⟶	*Matches:*
<atom> | (kwote <atom>)	any expression eq? to a
| (and <pat$_1$ > ... <pat$_n$ >)	if all of <pat$_i$ > match
| (or <pat> ... <pat$_n$ >)	if any of <pat$_1$ > through <pat$_n$ >match
| (not <pat>)	if <pat> doesn't match
| (? <predicate>)	if <predicate> is true
| (<pat$_1$ > ...[a] <pat$_n$ >)	a list of n elements
| <pat> ...[b]	a (possibly empty) repetition of <pat> in a list.
| #(<pat> ... <pat$_n$ >)	a vector of n elements
| #{<struct> (<field><pat>) ...}	a <struct>
| #{(<field><pat>) ...}	a <struct>
| ?<identifier>	anything, and binds *identifier* as a variable
| ?-	anything
| ??-	any (possibly empty) repetition of anything in a list
| ???-	any end of list

[a] Here, ... is a meta-character denoting a finite repetition of patterns.
[b] Here, ... means the special keyword "...".

Figure 1: Pattern Syntax

6 Error handling

There is no error procedure defined in the IEEE standard for the Scheme Programming Language. Bigloo has one and also a way of handling errors. This is achieved by two forms.

(**error** *proc msg obj*) bigloo procedure
This form signals an error by calling the current error handler with *proc*, *msg* and *obj* as arguments.

(**try** *exp handler*) syntax
The *exp* is evaluated with the *handler*. The *handler* is a procedure of four arguments. The first argument of the *handler* is the continuation of the **try** expression. The other arguments are *proc, msg and obj*. Invoking the first argument will resume the error.
Here is a small example of error handling.

```
(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
```

Some library functions exist to help writing handlers:

(**notify-error** *proc msg obj*) bigloo procedure
Display on error port *proc*, *msg* and *obj*.
(**dump-lambda-stack** . *depth*) bigloo procedure
Display on error port an execution trace (if compiled with the **-g** option) of size *depth*.

7 Assertions

Assertions allow to check predicates at certain points of programs. They can be enabled or disabled using Bigloo's compilation flags (**-g3** flag to enable them).
There are three ways to introduce assertions, depending on assertion scope.

(**assert entering** *exp* ...) bigloo syntax
(**assert entering** (*fn*) *exp* ...) bigloo syntax
The argument *exp* is a Scheme expression which has to be evaluated to **#t** when entering the function named *fn*. If *fn* is omitted, the assertion concerns the current defined function. This kind of assertion has to be placed in the program between function definitions and any other constructions which introduce bindings (e.g. **let**, lisp labels, ...). If the assertion failed (i.e. if *exp* is evaluated to **#f**), a **repl** is launched (see section 12.3) where the formal arguments of *fn* are bound to their actual values.

(**assert exiting** *exp* ...) bigloo syntax
(**assert exiting** (*fn*) *exp* ...) bigloo syntax
The argument *exp* is a Scheme expression which has to be evaluated to **#t** when exiting the function named *fn*. This kind of assertion has to be placed in the program between function definitions and any other constructions which introduce bindings (e.g. **let**, lisp labels, ...). If the assertion failed (i.e. if *exp* is evaluated to **#f**), a **repl** is launched (see section 12.3) where the formal arguments of *fn* are bound to their actual values. The result of *fn* is placed in the variable: ***assert-exiting-value***. This

variable can take place in *exp* and can be changed in `repl` if the assertion failed.

`*assert-exiting-value*` Bigloo variable
If a function contains an exiting assertion, it uses this variable to hold its results.

`(assert check (`*bdg* ...`)` *exp* ...`)` bigloo syntax
This form of assertion is the simplest. It means that *exp* has to be evaluated to `#t` at the point of the program where the assertion is placed. If the assertion failed, a `repl` is launched where the *bdg* variables are bound to their actual values.

Assertion forms are legal expressions which are *always* evaluated to the `unspecified` object.
Here is an example of assertion usage:

```
(module bar)

(define (foo x y)
   (assert entering (foo) (< x y))
   (assert exiting (foo) (> x 10))
   (let ((z x))
      (assert check (z) (>= z 0))
      z)
   (labels ((gee (t)
               (assert entering (gee)
                (>= t 0))
               (assert exiting (gee)
                (>= *assert-exiting-value* 10))
               (+ x t)))
      (set! x (gee y))))

(repl)
```

This module is compiled with the `-g3` flag to enable assertions, then the produced executable is run:

```
$ a.out

1:=> (foo 1 2)


*** ERROR:bigloo:assert:
assertion failed -- (ASSERT EXITING (GEE)
                     (2>= *ASSERT-EXITING-VALUE* 10))
---------------------------------
the values of the variables are:
   T : 2
---------------------------------
    0. FOO

*:=> ^D

*** ERROR:bigloo:assert:
assertion failed -- (ASSERT EXITING (FOO)
                     (2> X 10))
---------------------------------
the values of the variables are:
   X : 3
   Y : 2
---------------------------------

*:=> ^D

1:=> (foo 1 2)


*** ERROR:bigloo:assert:
assertion failed -- (ASSERT EXITING (GEE)
                     (2>= *ASSERT-EXITING-VALUE* 10))
---------------------------------
the values of the variables are:
   T : 2
---------------------------------
    0. FOO

*:=> (set! *ASSERT-EXITING-VALUE* 20)
#UNSPECIFIED

*:=> ^D

1:=> (foo -3 -4)


*** ERROR:bigloo:assert:
assertion failed -- (ASSERT ENTERING (FOO)
                     (2< X Y))
---------------------------------
the values of the variables are:
   X : -3
   Y : -4
---------------------------------

*:=> x
-3

*:=> y
-4

*:=> (set! x 40)
#UNSPECIFIED

*:=> ^D

*** ERROR:bigloo:assert:
assertion failed -- (ASSERT ENTERING (GEE)
                     (2>= T 0))
---------------------------------
the values of the variables are:
   T : -4
---------------------------------
    0. FOO

*:=> ^D
```

8 Structures

There is, in Bigloo a new class of objects: structures which are equivalent to C `struct`.

(`define-struct` *name field*$_1$ *field*$_2$ ...) bigloo syntax
This form defines a structure of name *name*, which is a symbol, having fields *field*$_1$ *field*$_2$ ... which are also symbols. This form creates several functions: creation, predicate, accessor and assigner functions. The name of each function is built in the following way:

Creator → `make-`*name*
Creator → *name*
Predicate → *name*?
Accessor → *name*-*field*$_i$
Assigner → *name*-*field*$_i$`-set!`

Functions `make-`*name* accept an optional argument. If provided, all the slots of the created structures are filled with it. The creator named `name` accepts as many argument as the number of slots of the structure. This function allocates a structure and fills each of its slot with its corresponding argument.

9 The foreign interface

We call foreign interface all the pieces of program devoted to the interactions between Scheme and other languages. In Bigloo the foreign interface allows to export Scheme's functions and variables to a foreign language and to import foreign functions and variables into the Scheme code. Using the foreign interface requires two kind of operations.

Declarations which are type declarations, import declarations or export declarations.

Foreign reference in the Scheme code.

Declarations take place in a special module clause (see 3). Using foreign variable in the Scheme code does not require any special construction.

9.1 The syntax of the foreign declarations

First, we give the syntax of the foreign interface.

```
<foreign> ⟶ <vclause>
        | <fclause>
        | <mclause>
        | <tclause>
        | <iclause>
        | <eclause>
```

variables The <vclause> describes the syntax for importing variables from a foreign language.

```
<vclause> ⟶
     ( <c-id> <variable> <c-name> )
```

The symbol <c-id> is the type's identifier of the foreign variable, <variable> is the identifier of under which the variable will be used in the Bigloo code and <c-name> is a string containing the real foreign variable name (its name in the foreign program).

functions The <fclause> describes the syntax for importing functions from a foreign language.

```
<fclause> ⟶
     (<c-id> <variable> (<c-id>*) <c-name> )
     | (<c-id> <variable> (<c-id>+ . <c-id>)
      <c-name> )
```

The symbol <c-id> is the type's of the result of the function. The symbol <variable> is the Bigloo's function identifier. The symbols <c-id>* are the type of the formal parameters of the function and the string <c-name> is the foreign name of the function.

macros The <mclause> is very close to the first two clauses given below.

```
<mclause> ⟶
     (define<c-id> <variable> <c-name> )
     | (define<c-id> <variable> (<c-id>*)
      <c-name> )
     | (define<c-id> <variable> (<c-id>+ . <c-id>)
      <c-name> )
```

These two clauses have the same meaning as the <vclause> and <fclause> ones. The only reason of using this <mclause> clause rather than the two first ones is that Bigloo won't generate (in the produced C code) any extern prototype for macros.

type This clause gives the syntax for defining foreign types. These types are usable in the foreign clauses. They allow the definitions of all the usual imperative language types.

```
<tclause> ⟶
     (type <c-id> <type-def>)
<type-def> ⟶ <atomic-type>
     | <c-id>
     | <struct-type>
     | <union-type>
     | <function-type>
     | <array-type>
     | <uarray-type>
     | <pointer-type>
     | <bitfield-type>
```

Let us detail how types are constructed.

atomic type The atomic types are the pre-existing ones. All the type prefixed by the '`b`' letter and the type `obj` are Bigloo's types. Other ones can be viewed as C types.

```
<atomic-type> ⟶ int | long | ulong | short
      | ushort | string | double | char
      | uchar | obj | bool | file
      | foreign | void | bint
      | breal | bbool | bpair | bstring
      | bchar | bprocedure | bsymbol
      | bvector | bstruct | binput-port
      | boutput-port
```

struct type This clause gives the syntax for defining foreign structure types.

```
<struct-type> ⟶
      (struct ((<c-id> <string>) ...)
       <string>)
```

All the <c-id> are Bigloo's type identifiers, <string> are the name of the foreign slots of the structure. The last <string> is the name of the foreign structure.

union type This clause is very close to the previous one.

```
<union-type> ⟶
      (union ((<c-id> <string>) ...)
       <string>)
```

function type It is possible to define type about function.

```
<function-type> ⟶
      (function <c-id> <formals>)
<formals> ⟶ (<c-id>* )
      | <c-id>
      | (<c-id>+ . <c-id>)
```

array type Two array types exist, the bound and unbound array types.

```
<array-type> ⟶
      (array <c-id> <integer> <integer>)
<array-type> ⟶
      (array <c-id>)
```

pointer type It is possible to define types which are *pointer-on* type.

```
<pointer-type> ⟶
      (pointer <c-id>)
```

bitfield type This type allows bit manipulations.

```
<bitfield-type> ⟶
      (bitfield <integer> <c-id> <string>)
```

export The <eclause> allows to tell to the compiler that some functions or variables will be used from the foreign code.

```
<mclause> ⟶
      (export <c-id> <variable> <c-name> )
      | (export <c-id> <variable> (<c-id>*)
       <c-name> )
```

The symbols <c-id> describe the type expected by the foreign language. The symbol <variable> is the Bigloo's identifier and the string <c-name> is the name for the foreign language.

include This clause forces the compiler to generate 'include' sentences into the produced code.

```
<iclause> ⟶
      | (include <c-name> )
```

9.2 The importations

The use of imported foreign functions or variables is straightforward. Programs can directly invoke or use them. When the compiler has found a usage of a foreign variable or when it has found an invokation of a foreign function, it compiles an implicit coercion. All types can be coerced. The figure below specifies if the coercion is allowed from Bigloo to foreign or from foreign to Bigloo or, in both directions.

bigloo	foreign	direction
bint	int	↔
breal	double	↔
bchar	char	↔
bstring	string	↔
bprocedure	function	←
output-port	FILE	↔
bbool	bool	↔

Furthermore, the foreign type `foreign` can be coerced with all Bigloo's types.

9.3 The exportations

It could be very useful to export functions (or variables) for a foreign language. Suppose a Bigloo program which implements the `fibonacci` function. Suppose that it is wanted to use this Bigloo's function in a C code. Since C's integers are not Bigloo's integers, two coercions are required, one before calling `fibonacci` and one after returning. The exportation protocol makes these coercions needless. The only requirement is to tell Bigloo's that some of its functions have to be exported for a foreign purpose.

Note: If a Scheme function appears in a foreign clause, it has also to be exported using a traditional Scheme export clause.

9.4 Creations and uses of foreign objects

For each foreign type defined in the foreign clauses, Bigloo creates functions to manipulate foreign objects. The way these functions are built is close to the way in which functions for manipulating `structure` are made.

(struct *name* ...**) :** This describes the way functions for manipulating `struct` are created.

Creator → `make-`*name*
Predicate → *name*?
Accessor → *name*-*field*$_i$
Assigner → *name*-*field*$_i$`-set!`

(union *name* ...**) :** The `union` objects are very close to the `struct` objects.

Creator → `make-`*name*
Predicate → *name*?
Accessor → *name*-*field*$_i$
Assigner → *name*-*field*$_i$`-set!`

(array *name* ...**) :**

(uarray *name***) :** The only differences between `array` and `uarray` are that, the functions for accessing or setting elements are safe (they test the range before accessing) with `array` and are not with `uarray`. Furthermore, the creation function for the `uarray` except one argument which is the length of the array to build.

Creator → `make-`*name*
Predicate → *name*?
Accessor → *name*`-ref`
Assigner → *name*`-set!`

(pointer *name* ...**) :** Bigloo creates functions for pointers.

Creator → `make-`*name*
Predicate → *name*?
Accessor → *name*`-ref`
Assigner → *name*`-set!`

Furthermore, some library functions help the usage of foreign objects. They are :
`foreign-null?` *obj* bigloo procedure
Returns `#t` if the object is a null foreign object.

9.5 The special case of structures and unions

Structures and unions are special. Each time Bigloo has found a type definition which concerns a structure (or a union), it automatically defines a type *pointer-on*. The name of this type is the name of the structure (or the union) followed by the character '`*`'. Implicit coercions are allowed from structure to pointer on structure (respectively union).

9.6 An example of foreign application

Let us see a small example that shows the usage of the foreign interface. First, we write a small C program :

```
#include "el.h"

int var = 9;

bar( x )
int x;
{
   return fib( x );
}

char *
hux( s )
char *s;
{
   static char string[ 500 ];

   strcpy( string, "TOTO IS HAPPY" );
   return string;
}

int
sum_el( head )
struct el *head;
{
int val = 0;

while( head )
   val += head->key, head = head->next;

return val;
}
```

Here is the "el.h" file:

```
struct el {
   int        key;
   struct el* next;
};
```

Now, we write a Bigloo module named "big-file":

```
(module big-file
   (foreign (int bar (int) "bar"))
   (export  (inline bis x)))

(define-inline (bis x)
   (bar x))
```

And, at the end, another Bigloo module which imports "big-file":

```
(module foreign
   (import  (bis big-file "big-file.scm")
            (main "main.scm"))
   (export  (test-foreign))
   (foreign (include "struct.h")
            (type el (struct ((int "key")
                              (el "next"))
                             "struct el"))
            (type pel (pointer el))
            (int sum-el (pel) "sum_el")
            (string hux (string) "hux")
            (int var "var")
            (int printf (string . foreign) "printf")
            (export int fib (int) "fib")))

(define (fib x)
   (if (< x 2)
       1
       (+ (fib (- x 1)) (fib (- x 2)))))

(define (foo x)
   (bar x))

(define (boo s)
   (hux s))

(define (test-struct n)
   (printf #"I'm goind to test: %d\n" n)
   (let ((head (make-el)))
      (el-key-set! head 0)
      (let loop ((n  n)
                 (c  head))
         (if (= n 0)
             c
             (let ((new (make-el)))
                (el-key-set!  new n)
                (el-next-set! new c)
                (loop (- n 1) new))))))

(define (test-foreign)
   (print var)
   (print (begin (set! var (+ 1 var)) var))
   (print (foo 4))
   (print (boo "toto is not happy"))
   (print (bis 5))
   (print (sum-el (test-struct 10))))
```

The successive outputs of this program are:

```
9
10
5
"TOTO IS HAPPY"
8
55
```

9.7 The very dangerous pragma Bigloo special form

Bigloo has a special form which allows the inclusion of C texts into the produced code.

(`pragma` *string*) bigloo syntax
Forces Bigloo to include in the produced C code the *string* as a regular C fragment code. This form must not be used without a smart understanding of the Bigloo C code production otherwise, the produced C file won't be compiled by the C compiler.

Remark: Other languages have such a feature, for example in C, this `pragma` syntax is often named `#asm` and allows the inclusion of assembler code in the C source file.

10 The extension package system

The extension package system is a way to extend the language compiled by Bigloo. It is achieved by associating an *extension file* and a suffix. The *extension file* is loaded at the beginning of a compilation. This file can call extern programs (unix programs), can modify the value of some variables of the compiler (for example, the list of the libraries to be linked with) and can define macros. The Bigloo's initializing procedure is the following:

1. Bigloo loads (if it exists) the runtime-command file (see section 13.4).
2. Bigloo parses the command line and finds the source file to compile.
3. Bigloo extracts the suffix of the source file and looks in its variable named `*auto-mode*`.
4. If the suffix is found, the associated file is loaded. This file could contain a function named `*extend-entry*`. This function must accept a list as argument. It is invoked with the Bigloo's unparsed arguments.
5. The result of the `*extend-entry*` application has to be a regular list of arguments. These arguments 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 package which is a front end compiler for the Caml language [Wei90].
Furthermore, Bigloo supports the `-extend` option (see 13.3) which forces the usage of an extension file. When

Bigloo encounters this option, it immediately loads the extension file then invokes the function `*extend-entry*` with the list of arguments which have not been parsed yet. The extension files are always searched in the directory containing the Bigloo's libraries.

10.1 User pass

Bigloo's allows the user to add special pass to the regular compilation. This pass take place *before* the macro expansion process. There are two ways to add a user pass.

Add a compiled pass: the module `user_user` (in the "comptime1.7/User/user.scm" file) is the user entry pass point. To add a compiled pass, put the code of the pass in this directory, imports your new modules in `user_user` and modify the `user-walk` function.

Add an interpreted pass: Set the value of `*user-pass*` in your `.biglooorc` file. This variable has to be a unary function. Bigloo will invoke it with the code as argument.

11 Macros

Two forms define user's compile-time macros:

`(define-expander` *name proc*`)` bigloo syntax
This form defines an expander named *name*, *proc* has to be 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.

User's expander are evaluated in an environment where only library procedures are bound.
The macro expansion process is done using expansion passing style [DFH86].

12 Miscellaneous

12.1 C requirement

The C code produced by Bigloo is ISO-C compliant [ISO90]. So, it is necessary to have an ISO-C compiler. The current version has been developed with `gcc` [Sta89].

12.2 Debugging tools

Debugging tools are really poor. They are two. The first is just an help by maintaining a call stack. When a runtime error occurs, this stack is then printed. The other tool is a special compilation mode which produces C code, which can be inspected with traditional debuggers, `gdb` [Sta88] for example.

12.3 The interpreter

It is allowed within the Bigloo's interpreter to call compiled code and to call interpreted code from compiled one.

`(eval` *exp*`)` bigloo procedure
This form evaluates *exp*.

`(repl)` bigloo procedure
This form invokes the *read-eval-print* loop. Several `repl` can be embedded.
`(quit)` bigloo procedure
Ends the current `repl` running. If the current `repl` is the first one then this function ends the interpreter.

`(h)` bigloo procedure
`(h` *n*`)` bigloo procedure
Displays the history list of the `repl` events; if *n* is given, displays only the *n* most recent events.

`(!` *n*`)` bigloo procedure
Re-evaluates the event number *n* of the history list.

`*prompt*` bigloo variable
This variable is a string which is the prompt used by the read-eval-print loop.

`(expand` *exp*`)` bigloo procedure
Returns the value of *exp* after all macro expansions.

`(expand-once` *exp*`)` bigloo procedure
Returns the value of *exp* after one macro expansion.
It is possible to specify files which have to be loaded when the interpreter is invoked, for this, see 13.4.
If a Bigloo file starts by 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.

12.4 Linking stand alone applications

Only Bigloo can be used to link object files which have been compiled by Bigloo. An easy way to perform this operation is, after having compiled all the files using the `-A` option, (see 13.3), to invoke Bigloo with the name of the compiled files.

12.5 Language restrictions

With a few exceptions, Bigloo implements the Scheme language. Some library functions differ as well as some other points, which are:

Arithmetic checking Bigloo does not generate any special checking for arithmetic overflow or underflow. You get whatever checking your C compiler gives you. We decided that the runtime checking was too expensive in a compiler that was constrained to produce C.

Arithmetic Only two arithmetics exist in Bigloo, `flonum` and `fixnum`.

Stack overflow checking Bigloo never checks stack overflows. Since C neither does, applications that overflow the stack will crash.

13 The compiler environment and options

There are three ways to change the behaviour of Bigloo. Flags on the command line, runtime-command file and environment variables. 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.

13.1 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. But, it is possible to suppress the genericity and to make all generic arithmetic procedures (= for example) fixnum ones (see also 13.3).

13.2 The safety

It is possible to generate *safe* or *unsafe* code (see also 13.3). The safety's scope is `type`, `arity` and `range`. Let's see an example:

```
(define (foo f vec indice)
   (car (f (vector-ref vec indice))))
```

In safe mode, the result of the compilation will be:

```
(define (foo f vec indice)
   (let ((pair
          (if (and (procedure? f)
                   ;; type check
                (= (procedure-arity f) 1))
                   ;; arity check
              (if (vector? vec)
                  ;; type check
                  (if (and (integer? k)
                           ;; type check
                           (>= k 0)
                           ;; range check
                           (<= k (vector-length
                                  vec)))
                           ;; range check
                      (f (vector-ref vec
                                     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 vec indice)
   (let ((pair (if (= (procedure-arity f) 1)
                   ;; arity check
                   (if (and (>= k 0)
                            ;; range check
                            (<= k (vector-length vec)))
                            ;; range check
                       (f (vector-ref vec indice))
                       (error ...))
                   (error ...))))
     (car pair)))
```

13.3 The Bigloo's command line

This section describes some of the arguments of Bigloo. To list all of them type under your shell:

```
$ bigloo -help
```

`bigloo [ -o` *output* `] [ -m` *module-name* `] [ -q ] [-I` *dir-name*`] [ -s ] [ -v ] [ -i ] [ -E ] [ -C ] [ -[cA] ] [ -g[`*23*`] ] [ -cg ] [ -unsafe[`*atrsv*`] ] [ -O[`*234*`] ] [ -farithmetic ] [ -w ] [ -copt` *args* `] [ -afile` *file-name* `] [ -access` *module-name file-name* `] [ -call/cc ] [ -mklib ] [ -mkheap ] [ -heap` *file-name* `] [ +heap` *file-name* `] [ -help ]` *input* `[ <-/+>rm ] [ -char <`*7/8*`>bit ] [ -nil ] [ -l`*library* `] [ -cc` *compiler* `] [ extend` *file* `] [ -query ] [ -version ]`

The different options are:

`-o` *output*	Name the output file *output*.
`-m`	Set the name of the current module. This means: overwrite the name in the module clause.
`-q`	Do not load an init file.

`-I`*dir-name*] Add *dir-name* to the load path

`-s` Compile silently.

`-v` Compile verbosly.

`-i` Interprete rather than compile the input files.

`-E` Stop compilation after the macro expansion.

`-C` Stop compilation before the C compilation.

`-[cA]` Don't invoke the linker, only produce a .o file.

`-g[`*23*`]` Produce some Scheme debugging informations. The several levels of debugging are:

- *1* force Bigloo to maintain an historic of global functions.
- *2* The historic is *also* maintained for local functions (this implies that tail recursion is not compiled without stack consumption).
- *3* Historic is maintained for global *and* local functions and assertions are enabled (see section 7).

`-cg` Produce some C debugging informations. This is useful to debug the executable with a C debugger, gdb for example.

`-unsafe[`*atrsv*] The compiler produces 'runtime safe test' to ensure the safety of the executable. It's possible to suppress several or all of these runtime tests. `-unsafe` suppresses all of them. The other flags are:

- *a* suppress checking arity.
- *t* suppress checking types.
- *r* suppress checking ranges (for vectors).
- *s* suppress checking structures.
- *v* suppress checking the compiler's soundness.

`-O[`*234*`]` 0ptimize the object code.

`-farithmetic` Don't use generic arithmetic operations. This means that the traditional operators will be `integer` operators. Nevertheless, it's still possible to use floating point numbers with the '`fl`' procedures.

`-w` Suppress warning messages.

`-copt` *args* Invoke the C compiler with *args*.

`-ld` *args* Invoke the linker with *args*.

`-l`*library* Link with object library *library*.

`-afile` *fname* Set the "access-file" to be named *fname*.

`-access` *m f* Add an access between module *m* and file *f*.

`-call/cc` Enable the library function "`call/cc`".

`-heap` *file-name* Use the file *file-name* as heap.

`+heap` *file-name* Use also the file *file-name* as heap.

`<-/+>rm` Remove or don't remove the temporary C file.

`-char <`*7/8*`>bit` Be eight bit clean or not. By default the char are 8 bits.

`-nil` Evaluates `'()` as `#f` in conditional expressions.

`-help` Give a brief description of the options.

`-cc` *compiler* Use *compiler* to compile C files.

`-extend` *file* Use the *file* as extension file. All the remaining arguments are parsed by the function `*extend-entry*` which could be defined in this file.

`-query` Dump the compiler configuration.

`-version` Give the Bigloo's version.

If no input file is specified, Bigloo enters its interpreter.

13.4 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 the exported variables can be found in the file: comptime1.7/Engine/param.scm.

Here is the list of the most useful ones:

`*cc*` A string which is the name of the C compiler.

`*indent*` A string which is the name of the C indenter.

`*cc-options*` Invoke `*cc*` with the string `*cc-options*`.

`*ld-options*` The linker options.

`*lib-dir*` The libraries path.

`*bigloo-user-lib*` A list which contains all the user's library.

`*bigloo-user-includes*` A list which contains all the user's include.

`*startup*` A string containing the name of file to be loaded before entering the "read eval print loop".

`*auto-mode*` An a-list (see section 10.).

`*user-pass*` User's pass entry point.

`*user-pass-name*` User's pass name.

The Bigloo's runtime command file is read before the arguments are parsed.

13.5 The environment variables

There are two kind of environment variables, the compile-time variables and the runtime variables. The first ones are:

`BIGLOOINCLUDE` The include path.

`BIGLOOLIB` The lib path.

`TMPDIR` The temporary directory.

The only runtime variable is:

`BIGLOOHEAP` The size in MegaBytes of the heap. If this variable is not bound, Bigloo will allocate a 4 MegaBytes heap.

References

[11791] IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[Bar89] Joel F. Bartlett. Scheme->c a portable scheme-to-c compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California, January 1989.

[Boe91] H.J. Boehm. Space efficient conservative garbage collection. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 28, 6*, pages 197–206, 1991.

[DFH86] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: Beyond conventional macros. In *Conference Record of the 1986 ACM Conference on Lisp and Functional Programming*, pages 143–150, 1986.

[ISO90] 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.

[QG92] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *Workshop on Static Analysis*, number 81-82 in bigre, pages 109–117, Bordeaux (France), September 1992.

[QP91] Christian Queinnec and Julian Padget. A Proposal for a Modular Lisp with Macros and Dynamic Evaluation. In *Journées de Travail sur l'Analyse Statique en Programmation Équationnelle, Fonctionnelle et Logique*, pages 1–8, Bordeaux (France), October 1991. Bigre 74 (oct 91).

[Sén91] Nitsan Séniak. *Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels*. Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), Paris (France), October 1991.

[Sta88] R.M. Stallman. Gdb manual. Technical Report Second Edition, Free Software Foundation, Inc., February 1988.

[Sta89] R.M. Stallman. Using and porting gnu cc. Technical report, Free Software Foundation, Inc., April 1989.

[Ste90] Guy L. Steele, Jr. *Common Lisp, the Language*. Digital Press, Burlington MA (USA), 2nd edition edition, 1990.

[WD93] Andrew K. Wright and Bruce F. Duba. Pattern matching for scheme. Technical Report ???, Department of Computer Science, Rice University, October 1993.

[Wei90] P. Weis. The CAML Reference manual, Version 2.6.1. Technical Report 121, INRIA-Rocquencourt, 1990.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399