23 things I know about modules for Scheme

Christian Queinnec
Université Paris 6 — Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex — France
Christian.Queinnec@lip6.fr

ABSTRACT

The benefits of modularization are well known. However, modules are not standard in Scheme. This paper accompanies an invited talk at the Scheme Workshop 2002 on the current state of modules for Scheme. Implementation is not addressed, only linguistic features are covered.

Cave lector, this paper only reflects my own and instantaneous biases!

1. MODULES

The benefits of modularization within conventional languages are well known. Modules dissociate interfaces and implementations; they allow separate compilation (or at least independent compilation *à la* C). Modules tend to favor re-usability, common libraries and cross language linkage.

Modules discipline name spaces with explicit names exposure, hiding or renaming. Quite often, they also offer qualified naming. These name spaces may cover variables, functions, types, classes, modules, etc.

Just as components, modules may explicit their dependences that is, the other modules they require in order to work properly. Building a complete executable is done via modules linking or module synthesis in case of higher-order modules. Modules dependencies usually form a DAG but mutually dependent modules are sometimes supported.

Proposals for modules for Scheme wildly differ among them (as will be seen) but they usually share some of the following features:

Determinization of the building of modules — For us, this is the main feature that allows users to build a system *S* exactly as it should stand, that is, without any interference of the current system where *S* is developped and/or compiled. This is in contrast with, say the Smalltalk way, where the state of the entire (development) system staid in memory (or in image files) making notoriously difficult to deliver (or even rebuild) stand-alone systems.

Interfaces as collection of names — If modules are about sharing, what should be shared ? Values, locations (that is variables), types, classes (and their cortège of accessors, constructors and predicates) ?

The usual answer in Scheme is to share locations with (quite often) two additional properties: *(i)* these locations can only be mutated from the body of their defining modules (this favors block compilation), *(ii)* they should hold functions (and this should be statically (and easily) discoverable). This restricts linking with other (foreign) languages that may export locations holding non-functional data (the `errno` location for instance). This is not a big restriction since modern interfaces (Corba for example) tend to exclusively use functions (or methods). On the good side, this restriction allows for better compilation since non mutated exported functions may be directly invoked or inlined.

Let us remark that values, if staying in an entirely Scheme world, would be sufficient since closures are values giving access to locations (boxes for dialects offering them will equally serve). Since values may be shared via λ-applications, module linking would then be performed by λ-applications without the need for, say, first-class environments [10].

Creating a namespace — A module confines all the global variables defined within its body. This global environment is initially stuffed with locations imported from required modules. Some directives exist to specify the exported locations. A location is specified by the name of its associated variable though renaming (at export or import) often exists.

To use locations is very different from the COMMON LISP way that rather shares symbols, with a read-time resolution, assigning symbols to packages. The Scheme standard is mute with respect to read-time evaluation or macro-characters that both heavily depend on the state of the system while `read`-ing.

Explicitation of required modules — In order to ease the building of large systems, modules should automatically keep track of their dependencies so requiring a module would trigger the requisition of all the other modules it depends on.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

Another personal word: I have ported Meroon for years on 12 different Scheme systems [7] and [8, p. 333]. While missing library functions (`last-pair` for instance) or obsolete signatures (binary only `apply` for instance) were always easily accomodated, the most problematic points had always been *(i)* the understanding of how macro-expansion and file compilation interfere and, *(ii)* how to install macros (`define-class` and related) into an REP loop. These problems were often solved by macros or invocations to `eval` thus introducing a new problem: the relationship between `eval` and macros!

Were not for macros, modules would probably be standard in Scheme for a long time. Alas! Macros or syntaxes extend Lisp or Scheme into new enriched languages providing syntactic abstractions that allows programmers to define abbreviations that simplifies the expression of how to solve problems. In mathematics, the "magic of (good) notations" has always transformed delicate semantical problems into syntactical routines (compare Euclide's elements language with usual algebra language). I tend to think that macros are probably the best reason for the survival of the Lisp family but there are the root of the problems for modules!

A module is written in some Scheme extended with various macros. It is expanded into core Scheme before being compiled. Macros do often occur in the module itself to be used in the rest of the module. This clearly requires the expansion engine to convert dynamically the definition of the macro (a text) into an expander (a function): this is the rôle of `eval` and the question is: what is the language used to define macros ? This language is another instace of Scheme possibly enriched with its own various macros. Therefore, in order to understand a module, a "syntax tower" or "macro-expansion tower" [9] must be erected. A module is therefore a tapestry of woven computations performed at different times within different variants of Scheme.

To sum up, a module proposal should solve many problems at the same time among which: share locations, manage their names, determine the exact language a module is written with, maintain module dependencies (for locations and languages) and, in case some invocations to `eval` appear in the built system, what language(s) do they offer?

2. TAXONOMY

In this Section, I will shallowly describe some features of some existing systems, I will then try to establish a rough taxonomy. This Section borrows some material from [8, p. 311]. I will use the term "abbreviation" for macros and syntaxes indifferently while I will only use syntaxes for R5RS hygienic syntaxes.

In the snippets, the "languages" in which they are written appear in right-aligned boxes. A language such as *Scheme+m1* means that the language is Scheme plus the `m1` abbreviation.

The classified systems are Bigloo [13], ChezScheme [14], Gambit [3], MzScheme [4] and Scheme48 [6].

2.1 Gambit

Gambit is probably the easiest to describe since there are no modules per se! Gambit [3] is centered on a REP loop; the predefined library offers the `compile-file` function compiling Scheme to C files that may be further compiled and linked with C means. A `(declare ...)` special form exists to alter the compilation behavior.

The language processed by the REP loop is assumed to be the one in which the file to be compiled is written. When an abbreviation is globally defined, it is immediately available. A global abbreviation defined while compiling a file is only available while compiling the rest of the file. Local abbreviations may be defined along with internal definitions.

Let us give an example of the various possibilities. The first snippet is performed within a first REP loop (whose prompt is `REP1>`). The snippet defines a function `f1` and an abbreviation `m1`: both are immediately usable in the REP loop and in the file to be compiled. The `m1` abbreviation is written in Scheme and may use `f1` at expansion-time. Uses of `m1` may be expanded into invocations to `f1`.

```
REP1> (define (f1 _) _)                          Scheme
REP1> (define-macro (m1 _)
          ;;(f1 _) is OK
         )
        ;;;(f1 _) and (m1 _) are OK               Scheme+m1
REP1> (compile-file "f.scm")
        ;;;(f2 _) and (m2 _) are not OK           Scheme+m1
```

Here is the content of the file, *f.scm*, to be compiled. It defines a function `f2` that may use the `m1` abbreviation (and its expansion-time resource `f1`). Invocations to `f1` and `f2` are of course allowed. Another abbreviation, `m2`, is defined whose scope (though "global") is only the rest of the *f.scm* file. Eventually a function, named `compute`, is defined wrapping a call to `eval`. The result is a compiled module that may require `f1` to run but always provide `f2` to whom will load it.

```
;;;File "f.scm"
(define (f2 _)                                 (Scheme+m1)
  ;;(m1 _) and (f1 _) and (f2 _) are OK
  )
(define-macro (m2 _)
  ;;(m1 _) and (f1 _) are OK
  )                                          (Scheme+m1)+m2
;;; (m1 _) and (m2 _) and (f1 _) and (f2 _) are OK
(define (compute exp)
  (eval exp) )
```

The language in which is written the *f.scm* file is not defined per se but due to the ambiant language in which `compile-file` is called, it is (scheme+m1). In the absence of compilation, the file just specifies that the `m2` abbreviation extends an unknown language. When compiled with the above conditions, the `m2` abbreviation is considered to be written in Scheme+m1. The body may also invoke `f1` which is indeed present at *f.scm* expansion-time.

The third snippet is run through another REP loop. The compiled *f.scm* file is loaded (a warning will be emitted to mention the absence of `f1`) then a function `f1` is defined (it might not be the same as the previous one defined in REP1) that will be used thoughout the rest of the REP loop. The language of this REP loop is Scheme without any abbreviation. The language accepted by the call to `eval` within the `compute` function is the current language in the current global environment.

```
REP2> (load "f")                                   Scheme
         ;;; (f2 _) is OK
REP2> (define (f1 _) _)
         ;;; (f1 _) and (f2 _) are OK
         ;;; (m1 _) and (m2 _) are not OK
REP2> (compute '(list (f1 _) (f2 _))) ; is OK
```

If REP2 were in fact REP1, `f2` would be loaded as before and `f1` would be redefined, the initial language would be Scheme+m1 instead of raw Scheme and the abbreviation `m1` would be allowed in `compute`.

Repeatibility of compilation is achieved by starting fresh REP loops. The model is simple, there is a single name space. No module dependency is explicit, missing or conflicting locations are caught by the compiler. The initialization order may be specified to the compiler.

The space of names is structured via namespaces offering the possibility of qualified names. A variable may be prefixed by the name of the namespace containing it therefore `m#f` is the variable `f` from namespace `m`. A `##namespace` directive rules, in the current scope, to which namespaces belong the defined variables.

2.2 Bigloo

Bigloo is compiler-centric. The compiler only compiles a single module i.e., some files with a prepended `module` clause. The `module` clause specifies the name of the module as well as some compilation directives. The rest of the file(s) is the source to compile (other files may also be adjoined with the `include` module directive.

The Bigloo compiler creates *.o* files (through C) or *.class* files for Java. When a module is mentioned in some module directives, the module is associated to at least one file and its `module` clause is read. Except when processing inlined exportations, the rest of the module is not read, that is, the `module` clause contains everything needed to compile or import it.

Expansion is performed with (EPS-style [2]) macros and/or (hygienically) with syntaxes. When a global abbreviation is defined, the compiler makes it available for the rest of the module. The language is which the module is written is specified by the `module` clause as well as its imported global environment.

Let us give a first, simple, example of a module, named `M1`. It only exports the immutable unary `f1` function (the arity and the immutability are implied by the shape of the `export` directive). It also defines an abbreviation `m1` whose definition is written in Scheme with the default global environment. This `m1` abbreviation may be used throughout the rest of the module.

```
(module M1 (export (f1 o)))

(define (f1 x) _)                                  Scheme

(define-macro (m1 _)
  ;; (cons _), (car _) are OK
  ;; (f1 _) is not OK
  )                                             Scheme+m1
;;; (m1 _) is now OK
```

Let us define a second module, named `f`. It exports the `f2` mutable variable (this is implied by the `export` directive) as well as the immutable unary function `compute` (that embeds a call to `eval`) and the immutable nullary function `get-bar`. The body of module `f` defines a global (that is, until the end of the module) abbreviation `m2`.

```
(module f
  (export f2
          (compute x)
          (get-bar) )
  (load (M1 "m1.scm"))
  (import (f1 M1 "m1.scm"))
  (eval (export f2)
        (import bar) ) )

;;; (f1 _) is OK                                Scheme+m1

(define (f2 _) _)

(define-macro (m2 _)
  ;; (m1 _) and (f1 _) are OK
  )
;;;                                          Scheme+m1+m2

(define (compute exp)
  (eval exp) )

(define (get-bar)
  bar )
```

The `load` clause of the `module` directive instructs the compiler to load the *m1.scm* file (not the `M1` module) therefore the `f1` function and the `m1` abbreviation are available to the compiler. The body of the `f` module may make use of the `m1` abbreviation and the expansion of an `(m1 _)` abbreviation may call `f1`. However if the result of the expansion contains a call to `f1`, the compiler would not find it in the global environment of `f` and would therefore warn the user. To fix this, `f1` is explicitly imported with another module directive. This directive only `imports f1`, this is to show that importation may import all or an explicitly named subset of the global variables of a module.

The last clause, the `eval` clause, specifies that `f2` will be made available to the language processed by the call to `eval` within `compute`. Conversely, it also says that the `bar` variable of `eval` may be used as the `bar` global variable within module `f`.

Let us now give a third module, named `M2`.

```
(module M2
  (import (f "f.scm"))
  (main start)
  (eval (export f1)) )

;;;                                                Scheme

(define (start arglist)
  ;; f2, compute, get-bar are OK
  _ )

(define (f1 x)
  (list "f1@m2" x) )
```

This is a main module whose entry point is the `start` function. This function may use the functions imported from module `f`. A third module directive exports for `eval` the current `f1` function defined in the current `M2` module.

When the whole application is started, a call to `compute` will use Scheme as language in a global environment made of `f2` (from `f`), `f1` (from `M2`) and `bar` (seen from `f2`). This language may evolve if enriched with new abbreviations submitted via `compute`.

The language of module directives is rich. It specifies importation, exportation (but no renaming) and re-exportation. Repeatability is ensured since only one module is compiled at a time: abbreviations cannot share state between compilations. The language of abbreviations may be specified (in Scheme but not in terms of compiled modules). The language of (all occurrences of) `eval` may be specified as well.

2.3 Scheme48

Scheme48 compiles modules in memory. An application is built by dumping the current state of the heap (one may also specify the function to invoke first when the image is resumed). The initial image contains the byte-code compiler and offers a REP loop able to interpret Scheme expressions as well as commands to inspect values or specify the module within which interpretation is performed. Commands are recognized by their leading comma.

Abbreviations are defined as specified in R5RS. Syntaxes are available immediately after being defined throughout the rest of the module.

Modules are built with a `define-structure` form (the name comes from SML terminology since it is possible (but undocumented in [6]) to define higher order modules). Modules export names (locations or syntaxes). There are some possibilities to filter the names to export as well as to modify them (both locations and syntaxes).

Our first attempt will define a first module, named `M1`, defining and exporting a function `f1` and an abbreviation `m1`.

```
;;; Within file m1.scm
(define (f1 _) _)
                                              Scheme
(define-syntax m1
  (syntax-rules ()
    ((m1 x) (list "m1@m1" (f1 x))) ) )
                                           Scheme+m1
```

After going in the `config` module, the `M1` module is compiled, at the level of the REP loop, with:

```
,config
(define-structure M1
  (export f1 (m1 :syntax))
  (open scheme)
  (files "m1.scm") )
```

The `M1` module imports the `scheme` module to gain access to the associated global environment (for example, for `list`) and syntaxes (for instance, for `define-syntax`). This double-sided importation is easily specified with `(open scheme)`. On exportation-side, the `m1` abbreviation is exported with the `:syntax` type. Due to hygien, the `m1` syntax captured the location of the `f1` function.

This first module may be imported by another module, `f`, whose body is contained in the *f.scm* file. This second module is compiled as follows (where the `m1` syntax is renamed `mone`):

```
(define-structure f
  (export f2 compute)
  (open scheme
        f (modify M1 (rename (m1 mone))))
  (files "f.scm") )
```

And its content is:

```
;;; Content of file f.scm
                                         Scheme+mone
(define (f2 _) _)

(define-syntax m2
  (syntax-rules ()
    ((m2 x)
     (list "m2@f" (f2 x)
           (mone x) (f1 x) ) ) ) )
                                      Scheme+mone+m2
(define (compute exp)
  (eval exp
        (scheme-report-environment 5) ) )
```

Due to hygien, the macro `m2` captures `f2` and `f1` but it also captures `mone`. The language in which are written expanders is Scheme which happened to define `syntax-rules`. Were we to use another language, we may enrich it with help of the `for-syntax` clause. Here is a variation of module `f` where `m1` is available to define the `m2` macro while the `mone` macro may only appear in the expansion of `m2`. The example is a little contorted since the use of `m1` is very gratuitous but it shows that Scheme48 differentiates the language of the module from the language in which syntaxes are written. This shows the first two level of the macro-expansion tower [9] named “syntax tower” in [6].

```
;;; Content of file ff.scm
                                         Scheme+mone
(define (f2 _) _)

(define-syntax m2
  (begin
    ;;                                     Scheme+m1
    (display (m1 111))(newline)
    (syntax-rules ()
      ((m2 x)
       (list "m2@f" (f2 x)
             (mone x) (f1 x) ) ) ) ) ) )
                                      Scheme+mone+m2
```

To compile the above module, we just open (that is, import), for the language of syntaxes, the `scheme` (for `display` and `newline`) and `M1` (for `m1`) modules:

```
(define-structure ff
  (export f2)
  (open scheme
        (modify M1 (rename (m1 mone))) )
  (for-syntax (open scheme M1))
  (files "ff.scm") )
```

Scheme48 compiles in memory so it offers various interesting effects: it is possible, at the REP loop, to place oneself in the context of a module to evaluate some code and even to enrich the current language and global environment:

```
,in F (list (f2 33) (m2 44) (m1 55))
,in F (define (f3) "f3@M1")
,in F (define-syntax m3
        (syntax-rules ()
          ((m3) (list (f2 (f3)))) ) )
,in F (m3)
```

The REP loop offers some features useful for development; for instance, it is possible to reload a module without changing the exportation contract.

Since all modules are known from the REP loop, there is no per se module dependencies. However to determinize the building of an image requires to be able to reset modules to their initial state (in order to reset syntaxes with shared state), something possible with the `reload-package` command.

Whereas the language of modules and syntaxes is well defined, I did not see any possibility to specify the language of `eval` when specifying the module. It is possible though in R5Rs with the usual `scheme-report-environment` function and the like; this is probably also possible making use of the internal `get-package` function.

2.4 Chez Scheme

This Section is only inferred from Waddell's and Dybvig's paper [14]. A module is alike a definition (it may appear wherever a definition may occur (globally or locally)) and looks like `(module` *`module-name`* `(`*`exported-names`*`)` *`body`*`)`. A module opens a new namespace that captures all definitions (variables or syntaxes) among which some are exported as mentioned by *exported-names*.

Free variables of the module are also captured by the `module` form but they are not exported. Such a form defines a kind of first-class environment named *module-name* except that syntaxes are also exported.

```
(let ((x 1))
  (module A (f)
    (define (f z) (list x z)) )
  (module B (g)
    (define (g y) (f y)) )
  (import A)
  (let ((x 2))
    (import B)
    (g x) ) )   ;yields (1 2)
```

Modules are imported with the `(import` *`module-name`*`)` form. This is again a definition form that may appear wherever a definition may occur. When an importation occurs locally the exported names participate to the `letrec` effect as the other internal definitions.

This module system is intimately tied with `syntax-case`: an interesting corollary is that a whole program making use of `module` and `import` forms is transformed, after macroexpansion, into a core Scheme expression (that is, without abbreviations or derived syntaxes). The `syntax-case` facility allows for selective importations, renaming individual variables and gathering exportations with the sole means of hygien (see [14, Section 3.3] for details). It does not seem to allow the renaming or prefixing of all exported variables.

Here are some (untested) examples though they do not make these modules to shine.

```
(module M1 (f1 m1)
  (define f1 _)
  (define-syntax m1 _)
  )

(let ()
  (import M1)
  (module F (f2 compute)
    (define f2 _)
    (define-syntax m2 _)
    (define (compute exp) (eval exp)) ) )
```

Good examples where modules are imported in a local scope are given in the paper [14] however, separate compilation of local modules does not seem practical. These modules do not seem to allow the specification of the language of expanders though the strict and sole use of `syntax-case` alleviates this need. Nor they allow the specification of the language of `eval`.

A very interesting property mentioned in [14] is the structure of the compiled module. Since a module may export locations or syntaxes, the compiled code contains the code to initialize the locations and the code for the exported expanders. When `visit`-ing a module only the expansion resource are set up while `load`-ing a module also initialize the regular locations. This might have been done, in plain old Lisp, with `eval-when`: the compiled code related to syntaxes is therefore conditionalized with a kind of `(eval-when (visit) ...)`.

2.5 MzScheme

MzScheme 200 is the most recently implemented module proposal [4]. It improves on Chez Scheme's module system and solves a number of problems.

A module form specifies its name (bound in a specific namespace), the language it is written in and its body (a sequence of definitions (locations and syntaxes), exportations, importations and expressions). Exportations (of locations or syntaxes) are specified with the `provide` form. This form offers facility to rename, prefix or selectively hide names.

Importing a module is performed with the `require` form; importations may also rename, prefix or selectively hide names. The importation brings in names of locations or syntaxes. Note that importations and exportations are not gathered in a single place.

Let us give an example of a module `M1` exporting a function and a syntax. The module is written in MzScheme; the language of the abbreviation is not specified but as abbreviations adopt the syntax language of R5RS, it should at least contain this latter.

```
(module M1 MzScheme
                                    MzScheme
  (define (f1 _) _)
  (provide f1)
  (define-syntax m1
    (syntax-case _) )
                                    MzScheme+m1
  (provide m1)
)
```

Here is a second module, `F`, that imports `M1` environment and syntax.

```
(module F MzScheme
  (provide f2 compute)
  (require M1)
                                    MzScheme+m1
  (define (f2 _) _)
  (require-for-syntax (rename M1 m1 mone))
  (begin-for-syntax
      ;;                             R5RS+mone
    (mone _) )
  (define-syntax m2
      ;;                             R5RS+mone
      _ )
                                  MzScheme+m1+m2
  (define (compute exp)
    (eval exp) )
)
```

Modules offer the usual syntax tower. In the `F` module, the language for syntaxes also imports `M1` (its function `f1` and syntax `m1` renamed `mone`) therefore, the language for syntaxes is R5RS enriched with `mone`. A specific syntax, `begin-for-syntax`, evaluates its body in the language of syntaxes (something not so dissimilar to `eval-in-abbre-viation-world` [9]). Let us focus a little on `begin-for-syntax`. Compare the old writing with plain old macros with the new syntax[1]:

```
(define-macro (foo _)
  (hack)
  `(bar _) )
```

is now written as

```
(define-syntax foo
   (syntax-case
     (_ (begin (begin-for-syntax (hack))
                 (bar _) )) ) )
```

Since dependencies are explicit, `require`-ing a module *M* recursively `requires` the modules *M* requires. Compiling a module *M* requires the modules *M* requires for syntax in order to initialize the syntax tower and its first level: the syntax language. Modules contains sequences of code associated with their phase (run-time, expansion-time, etc.) and only the needed part is run when required by a specific phase. Repeatability is ensured since modules' environments are not shared by differing phase: if a module is required at some phase, it will be reinitialized when required at a different phase.

Concerning explicit evaluation, there also exists in MzScheme namespaces to provide global environments for `eval` (the standard `scheme-report-environment` function creates namespaces). They do not seem to be associated with a syntax tower.

3. TAXONOMY

All these modules systems are very different, they have various goals and few common points. Here is an attempt to classify them.

What is a program? Scheme48 and MzScheme specify what is a program with a grammar defining and instantiating modules. ChezScheme proposes a transformation mapping a program using modules into a single S-expression. Bigloo and Gambit compile towards C (or JVM) and leave `ld` build programs.

Do modules support separate compilation? This sieves ChezScheme embeddable modules from the others.

Do modules support interactive debug? Debugging means, most of the time, violating the language (modifying a constant, conditionally aborting computations, etc.): debugging is not constrained by the language. Offering a toplevel for debug as in Scheme48 complexifies the semantics.

Do modules support classes? Classes are not defined by Scheme but all systems offer a variant of them. Bigloo is the only one that combines class definition and exportation.

4. VAGUE FEELINGS, FUTURE QUESTIONS

This Section is highly hypothetic, it only reflects some instantaneous feelings about modules and macros. It also contains some shallow ideas that need much, much, much work to be published :). Of course, readers are not compelled to share these feelings!

Modules do not need to be embeddable, top-level modules with explicit importations and exportations allow for easier separate compilation. I also tend to think that higher-order modules are not needed in an statically untyped language such as Scheme (generic functions are probably sufficient).

Specifying a language or a global environment are two different things that operate at different times with very different goals. Languages must be totally defined in order to expand modules: they extend the compiler with a pre-pass (an expansion pass). Therefore a language may be represented by a transformer that converts expressions using some abbreviations into expressions that do not use these abbreviations. Therefore an abbreviation may be seen as a language transformer that is, creating a new language enriched with a new abbreviation.

Today, the abbreviation protocol fuse all abbreviations into a single transformer. To stage transformations would be benefitful for instance for macros that want to code-walk expressions after transformation to core Scheme (so they are free of implementation-dependent special forms). How to compose abbreviations into passes and how to rank passes is open.

Constituting global environments is rather independent of the compiler. Even if requiring a unique thing, such as `scheme`, bringing both a global environment and a language is, of course, easier for the user, I tend to separate expansion and linking.

Importation language should allow arbitrary computations on sets of names (for instance, managing the whole set of names associated to a class definition, or names obeying a given naming pattern). The importation language should also be able to accompany sets of locations with extra informations required for better compilation. This extra information should not obfuscate importations.

The only operation that can be performed on a compiled module should be to load it (not to visit, import, use or whatever). I therefore favor a mode where a module is compiled

[1] I tend to think that the first one makes easier to understand the two different languages that are involved.

into a single, monolithic, that is, non conditionalized, code. However, compiling a module requires expanding its body. Expansion requires an evaluation that is done with an appropriate syntax tower. Compiling another module requires another appropriate syntax tower.

On the evaluation side, it should be possible to build specialized `eval`-uator(s) for any given language. Different parts of the whole executable may need more than one language for extension. It should also be possible (maybe with first-class environments [10]) to set up the needed sharing.

The synthetised `eval` takes an expression and a global environment as in R5RS. The returned evaluator comes with its own syntax tower, the language of the macros for this evaluator may be obtained. An example of these functions may be as follows:

(`make-eval` *language-expr*) → *evaluator*
(*evaluator expression environment*) → *value*
(`syntax-eval` *evaluator*) → *evaluator*

A *language-expr* is an expression in a language definition language, a naive example might be:

(*base-language macro* `...`)

Finally, language expressions may also be used to specify local languages to use:

(`with-language` *language-expr s-expr*)

Since a language is seen as an expression transformer it may be obtained by loading a module. Finally, repeatibility must be the paramount property of this system (with first-class languages?) to offer real separate compilation.

5. CONCLUSIONS

This paper discusses various points offered by some module systems for Scheme, some problems they solve or not and some ideas about them. As a conclusion, it seems highly hypothetic to add soon a chapter on modules in R^6RS. However, thinking positively, I propose two measures that should be simpler to introduce:

- Documentations should explain the syntax towers they use (for their toplevel, modules, `eval` or `expand` facilities).
- Introduce an `eval` function with an additional third argument specifying the syntax tower to use.

The source of the various experiments may be found via the net at:

`http://youpou.lip6.fr/queinnec/Programs/sws-2002Aug18.tgz`

6. REFERENCES

[1] P. Curtis and J. Rauen. A module system for Scheme. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[2] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *International journal on Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[3] M. Feeley. *Gambit-C, version 3.0 – A portable implementation of Scheme*, May 1998.

[4] M. Flatt. Composable and compilable macros: You want it *When?* In *ICFP '2002 – International Conference on Functional Programming*, Pittsburgh (Pennsylvania, US), Oct. 2002. ACM.

[5] R. Kelsey, W. Clinger, and J. Rees, editors. Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[6] R. Kelsey and J. Rees. *The Incomplete Scheme 48 Reference Manual for release 0.57*, 1999. with a chapter by Mike Sperber.

[7] C. Queinnec. Designing MEROON v3. In C. Rathke, J. Kopp, H. Hohl, and H. Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), Sept. 1993.

[8] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[9] C. Queinnec. Macroexpansion reflective tower. In G. Kiczales, editor, *Proceedings of the Reflection'96 Conference*, pages 93–104, San Francisco (California, USA), Apr. 1996.

[10] C. Queinnec and D. De Roure. Sharing code through first-class environments. In *Proceedings of ICFP'96 — ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia (Pennsylvania, USA), May 1996.

[11] C. Queinnec and J. Padget. A deterministic model for modules and macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[12] C. Queinnec and J. Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), Oct. 1991. Plenum Publishing Corporation, New York NY (USA).

[13] M. Serrano. *Bigloo — A "practical Scheme compiler" for Bigloo version 2.5b*, July 2002.

[14] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 203–213, New York, NY, 1999.