Modules, Macros and Lisp

Improved version of Technical Report 90-36,
University of Bath, United Kingdom.

Christian Queinnec[*]
École Polytechnique — INRIA
91128 Palaiseau CEDEX, France
queinnec@poly.polytechnique.fr

Julian Padget[†]
University of Bath,
BATH, Avon, United Kingdom
padget@maths.bath.ac.uk

may 11th, 1990 — EuLisp issue (Draft)

Abstract

Lisp has some specialised capability for reflective operations, exemplified by its macro facility, structured name spaces, file compilation, file loading and dynamic code synthesis. There has been some progress in the last few years on the semantics of macros, but the other operations have been categorized as environmental issues. In this paper, we present a semantics for modules and will show that it substantially reduces the difficulty of defining precisely several features of usual Lisp systems such as macros, module compilation (file compilation), module loading (fasl loading) and dynamic evaluation. The module schema addresses the questions of name visibility and separate compilation. Macro-expansion is specified relative to where and how it takes place as part of operations on modules.

We do not present a radical Lisp, but rather one that tries to stay within the commonly understood Lisp paradigm. We do not simply borrow from other languages — the particular behaviour of Lisp precludes this. The semantics we have developed describes a Lisp with modules and macros enhancing portability and understandability.

1 Goals

One of the main criticisms against Lisp is its inability to separate the language from the environment. Although, the success of Lisp is largely due to this confusion since it allows to easily build convenient programming environments. As a reaction, Scheme tends to focus on the language itself unembarrassed from non-linguistical features. We thought that the continuation of the formalization effort of the Scheme community should address the module aspect and we originally focused on it. Our vision was to provide a Lisp composed of a multitude of linked and reusable modules. It soon appeared that our formalization of modules greatly simplifies the way we perceive macros and `eval` feature. We thus believe that a module facility such as ours allows to clearly separate the intricated mess of syntactic and semantics problems that were lurking for long around.

This paper is made of three parts dealing respectively with modules, macros and evaluation. We first present an essential module facility which is inspired by the universal import/export mechanism. The formal semantics of these modules is thereafter commented since we believe that no complete module definition can be discussed without formalization. Many subtle issues (such as reloading a module or macros generating new macros) cannot be perceived in absence of such a formalization. On that basis the macro problems (which are numerous) are presented one after one with examples. Despiste being tightly coupled we tried to separate them into independent issues: this part may probably help Lisp programmers to master whatever macro machinery. The formal semantics of our macro model is then presented as a list of steps to be added to the module semantics. The price paid for macros makes easy to introduce the dynamic evalution facility. By the way this feature helps us to alleviate some of the limitations raised for macros.

[*]This work has been partially funded by Greco de Programmation.
[†]Part of this work was carried out whilst visiting INRIA (Rocquencourt).

In this paper we tried to respect the Lisp spirit and thus to limit the number of new constructs (there are four of them, namely **`defmodule`**, **`loadmodule`**, **`startmodule`** and **`with-syntax`**). Many usual features of Lisp find a way to be expressed in our model. These can be denotations (for the afore mentionned new constructs) or simply Lisp code (for instance **`local-defmacro`**) appearing in libraries. The resulting model achieves the original vision, retains the "power" of Lisp and satisfies both the implementor and the user since most of the system is written in Lisp.

2 Modules

This section describes the needed module facility and the usual solutions taken elsewhere. We then describe an essential solution with some examples. We finish the section by the formal denotational semantis of our module proposal.

2.1 Which Modules for Lisp ?

The modules we want must allow control over resource visibility and over separate compilation. Lisp's heavy reliance on latent types and thus the burden of checking at run-time poses some major problems for separate compilation. What we will describe is a separate compilation model based on the definition and linking of modules to build (stand-alone) applications[1].

For the purposes of our discussion, we separate the language into two levels: (i) the usual programming level, comprising special forms (for example, **`if`**, **`setq`**, **`lambda`** and **`quote`**), variable reference and combination and (ii) the module level comprising operations to define and control modules. To category (i) we add the special form **`loadmodule`**. In category (ii) we have two pseudo special forms: **`defmodule`** and **`start-module`**. We term these *pseudo* special forms since they are not necessarily part of the usual programming level and it is possible to think of several different styles of implementation. For example: modules could be defined and evaluated either from a toplevel loop or from some job control language (the UNIX[2]shell, for instance).

Resources are usually shared between modules using some variation of the import/export paradigm. Imports describe which objects defined outside a module are to be visible within it. Exports describe which objects defined by a module are to be visible outside it. There is an imbalance in the amount of information available about each of these. Whilst everything is known about the exports, much less is known about the imports. Because the knowledge of the imports is imperfect, some kind of checking is required on how an imported object is used. Such checks can be performed at compile-time, load-time or even at run-time depending on the kind of objects that are exchanged. If we restrict the exchange of information to concern only variables[3], three options are possible: exchange names (as in C or COMMON LISP), values (as in Poly[Matthews, 1983]) or bindings[4] (as in Ada[5]). We discuss each of these in more detail now:

names To exchange names implies that there exists a sort of single namespace since it is not possible to import two different resources with the same name. Needless to say, to share names leads to sharing all the properties they convey with them, but, principally, their value. In COMMON LISPfor instance, to know a name (a symbol in that world) gives access to its value, its functional value, its property list, its package, the other symbols of the package ... We are in agreement with Moon's assertion[Moon, 1988] that packages are not a module mechanism and should not be abused as such. Thus, we are not in favour of the exchange of names.

values To exchange just values is a kind of negation of modules since to export a value is to compute it, while importing a value is just to bind it under a local name. A valued-based module system is therefore like a closure.

bindings Bindings are not first class entities but they are independent of names, so bindings may be imported and renamed locally. Hence, we propose to investigate further a binding based model of modules.

[1]Throughout this paper *application* will denote a stand alone program rather than the case of applying a function to some arguments (which will be referred to as a *combination*).

[2]UNIX is a registered trademark of UNIX Systems Laboratories, Inc. in the USA and other countries.

[3]Since we have taken a Scheme-like dialect of Lisp as our base language, our language is single-valued and there is no difference between exporting a variable and exporting a function.

[4]The `var` directive of Pascal is a good example of how bindings can be shared.

[5]Ada is a trademark of the Ada Joint Program Office.

Efficient compilation is difficult. Inlining or suppressing the arity test cannot be done if functions are only known at run-time. To know types at compile-time and to exchange them may lessen these penalties. In a typed language, global variables are typed and so are their associated bindings. This type information can be kept as a property of the binding and therefore also exchanged in order to permit inter-module checking and optimization. For us, a binding is a location, perhaps accompanied by some useful information. Names can also convey at compile-time the same extra information but do not offer the opportunity for renaming. Values only convey their types but only at run-time and thus preclude any compile-time optimization. We are not proposing a typed Lisp here, we only stress, as it is currently done in some implementations, that some type informations (explicitely provided by the user or inferred by the processor) can be transferred via the import/export mechanism.

In summary, we think that modules are essential for modern Lisps and furthermore that modules do not correspond directly to closures. We also want to reduce the number of symbols, which as a side-effect consume a lot of memory, but also create the kinds of loophole we identified above. We therefore choose to share bindings — that is, locations.

2.2 An Essential Solution

The essence of the module scheme proposed here is sufficient to describe the abstract syntax and to keep the size of the denotational descriptions manageable. The model defined in this section is *only* for the purpose of discussing the meaning. This solution is sufficiently powerful to satisfy our needs. It is simply inconvenient for practical use. A proposed syntax of a practical model is given in section 2.7, but the principles are those of this section.

We define the syntax of the pseudo special form **`defmodule`** as follows:

defmodule	::=	(**`defmodule`** *name import-spec export-spec form*)
import-spec	::=	(*an-import-spec**)
export-spec	::=	(*name**)
an-import-spec	::=	(*module-name exported-name imported-name*)

We now examine the syntactic components of the **`defmodule`** form in detail:

name A module has a *name* which identifies it in the Module Environment: **ModEnv**.

import-spec The import specifications declare which bindings are to be imported from other modules. All imports are explicit — that is to say there are no implicit or automatic imports. However, the special forms of the language are available in every module, because they are part of the syntax of the language — they are not functions, indeed the names are not bound to anything, and therefore they cannot be imported. The import specification is a list of 3-tuples, each of which specifies the importation of a single binding. The elements of the 3-tuple name the module from which the binding should be imported, the exported name of the binding in that module and the name the binding is to be given in this module, respectively.

export-spec The export specification is a list of names identifying the bindings to be exported. Exported bindings must correspond to top-level bindings of the module — that is, bindings associated to names appearing free in *form*. Imported bindings can be re-exported providing restriction or encapsulation capabilities as well as partial-linking.

form The *form* is the body of the module[6]. This form will only be evaluated at module loading *not* at module definition. It suffices that the free variables can be extracted at definition time.

Let us give an example of a simple module: `(defmodule Ackermann`

```
((arithmetic integer-eqn =)       ;;Import =, - and <
 (arithmetic integer-subtract -)
 (arithmetic integer-lessp <))
(ack ack-max)                     ;;Export ack and ack-max
(progn
  (setq ack-max 4)
  (setq ack
    (lambda (x y)
      (if (= x 0) (- y -1)
```

[6] The restriction to a single form is intentional, since it avoids unnecessary complexity in the semantic equations. Besides, we assume the existence of `progn`.

```
              (if (= y 0) (ack (- x 1) 1)
                  (if (< x ack-max) (ack (- x 1) (ack x (- y 1)))
                      (error "Compute it yourself !")))))))
```

Two top-level bindings exist in the **Ackermann** module and both are exported. Thus, these bindings may be used in elsewhere. To use the **Ackermann** module a student may write: `(defmodule student`

```
  ((Ackermann ack ackermann)       ;; only import the function
   ...)                            ;; also import dotimes and printf
  ()                               ;; exports nothing
  (dotimes (i 31)
    (dotimes (j 31)
      (printf "Ackermann(~D,~D)=~D~%" i j (ackermann i j)))))
```

while a teacher may write: `(defmodule Ackermann-for-students`

```
  ((Ackermann ack ack)
   (Ackermann ack-max max))
  (ack)
  (setq max 3))
```

and require the students to use the new module **Ackermann-for-students** to limit cpu consumption. Note that when this module is loaded, the **Ackermann** module will also be loaded and the binding of its **ack-max** variable will be modified to `3`. The **Ackermann-for-students** is an encapsulation of the original **Ackermann** module, denies access to **ack-max** but transmits the **ack** function as it is. We leave access rights to modules themselves as an environmental issue.

So far this section, we have defined the syntax for our essential model and illustrated how this model can be used. In the following three subsections we give English semantics for module definition, loading and entry before turning, in the fourth succeeding subsection, to the denotational semantics of module operations.

2.3 Module Definition

When a module is defined, the following analysis takes place:

1. Importations are checked. For each imported binding (*module-name exported-name imported-name*), the module *module-name* must exist and *exported-name* must be exported from it. Each binding import specification contributes to the *top-level* environment being built for the module being defined. Each such binding is checked for name-conflict, since no two imported names can be the same. Note that mutually referential modules are not possible because of the definition before use requirement. Hence, the importation dependencies form a DAG.

2. The body of the module is analysed and all free variables are extracted. The free variables are added to the module's top-level which already contains the imported bindings. If such a variable is new, then it is associated with a new location. It is important to note that the extraction of free variables depends only on the knowledge of the set of special forms of the language. The special problems created by macros will be addressed in section 3.

3. Exportations are checked. Each name appearing in the export specification of the module must also be defined in its top-level environment.

4. The module is added to the module environment **ModEnv** bound to the given *name*.

All the previous steps are known as the *module definition* phase. Module definition is done in a null lexical context and corresponds roughly to the construction of an interface description. Note that, so far, no evaluation has been performed.

After a module has been defined, it can be used in the definition of other modules — that is, its bindings can be imported (and re-exported) — all without evaluating anything in the original module. We leave as an environmental issue the consequence of module redefinition: it perhaps involves recompiling all depending modules.

The exposed module scheme allows separate compilation in the following sense: modules may be compiled in whatever order provided this order satisfies the constraints imposed by the import clauses. This independence will be clear from the denotation of section 2.6. Our view of modules and their use in separate compilation completely dissociates the use of (reference to) a module from its evaluation (loading). Reference to a module provides information about its exported bindings and allows reference to them. It is loading a module that allows the *d*ereferencing of its exported bindings to gain access to their values. The act of loading a module causes its bindings to be filled in.

Another good point of the module scheme is the precise control one has over time which allows to reject semantical obscenities such as **`eval-when`** [Steele 90, Dybvig 87]. If one wants something to be processed at module definition time then macros (see later) are there, if one wants some code to be run at load time then this code must be included in the body of the module.

2.4 Module Loading

To dereference the bindings of a module, it must be loaded. A module is loaded by means of the **`load-module`** special form. `(load-module` *module-name*`)`

A module can be in one of two states: *loaded* or *not-loaded*. Regardless of the state of a module, the **`load-module`** form always loads *module-name*.

Loading a module causes the following sequence of actions to be taken:

1. All the modules imported by *module-name* are loaded — if they are not yet loaded. If they themselves import other modules, those modules will be imported first, and so on. The DAG describing import dependencies is traversed in order, loading modules as necessary. The result is that every imported binding will really exist somewhere in the store before any attempt is made to dereference it.

2. The proper part of the top-level environment is allocated. The top-level environment of a module is divided into two parts: its proper part, which defines the locations for the free variables defined in the body of the module and the imported part, which is the set of proper parts of the imported modules' top-level environments.

3. The body of the module is then evaluated in that newly-created top-level environment.

4. The module is marked as *loaded* to avoid automatic re-loading (see step 1).

5. The value of the **`load-module`** special form is the value returned by the evaluation of the body of the module.

A module can always be reloaded explicitly using **`load-module`**, in which case the process starts from step 3. Reloading a module does not require the reloading of the modules it imports since the exported locations do not change and similarly with modules that import it. The ability to load a module explicitly allows control over the order of module loading.

2.5 Module Entry

Having defined an application constructed from a collection of modules, some way is needed to start the application and provide some initial arguments. For this, we have the **`start-module`** pseudo special form. `(start-module` *module-name function-name OS-arguments*`)`

Any exported binding can be the entry point of a potential application provided that *module-name* is an existing module and that *function-name* is an exported binding. The *module-name* is then loaded and the contents of the *function-name* binding is retrieved, checked to be a function and then applied to the unevaluated *OS-arguments*. The *OS-arguments* represents the initial information given to the application: it may well be something which looks like `(argc, argv)` as in UNIX, but we purposely do not specify the format, at present. It is the duty of **`start-module`** to place these *OS-arguments* in the initial store of the application. The **`start-module`** form is evaluated as if in the null lexical context.

We can now explicitely state what is a program in our world. A program is a sequence of module definitions followed by a single **`start-module`** form. This contrasts with the usual interactive philosophy of Lisp where a toplevel loop allows the user to submit forms, function, module or macro definitions as well as she can load or compile files, disassemble or debug applications. In our wiew this toplevel loop is a particular application: a kind of *shell* which does not strictly implement the language since it allows to alter module meaning, violate information hiding, mutability of bindings and so forth. A module defining a toplevel loop will be discussed in section 4.2.

We claim that our semantics as it is[7] can be implemented by truly equivalent compiler or interpreter since it is the semantics of a language not of a particular application. We thus disconnect the language and the environment.

2.6 Module Semantics

We now turn to the question of the formal meaning of these module operations in the context of a simple Lisp-like language. This formalization is necessary since the definition of macros and eval features would strongly depends on it and, given the subtleties and pitfalls related to these concepts, a clean semantics is obviously needed. The domains of such a language are:

$\phi \in$	**Fun**	$=$	$\mathbf{Value}^* \times \mathbf{Store} \times \mathbf{LoadEnv} \times \mathbf{Cont} \rightarrow \mathbf{Answer}$
$\kappa \in$	**Cont**	$=$	$\mathbf{Value} \times \mathbf{Store} \times \mathbf{LoadEnv} \rightarrow \mathbf{Answer}$
$\rho \in$	**LexEnv**	$=$	$\mathbf{Id} \rightarrow \mathbf{Location} \cup \{\textit{lexenv-location-not-found}\}$
$\sigma \in$	**Store**	$=$	$\mathbf{Location} \rightarrow \mathbf{Value} \cup \{\textit{store-undefined-value}\}$
$\varepsilon \in$	**Value**	$=$	$\mathbf{Fun} + \mathbf{Sexp} + \ldots$
$\alpha \in$	**Location**		
	Meaning	$=$	$\mathbf{Id} \times \mathbf{ModEnv} \times \mathbf{LexEnv} \times \mathbf{Store} \times \mathbf{LoadEnv} \times \mathbf{Cont} \rightarrow \mathbf{Answer}$
$\nu \in$	**Id**		*an identifier*
$\pi \in$	**Form**		*a form*

Several notational conventions have been employed. These are taken from [Rees & Clinger 1986] except that we suffix variables with small letters to indicate their origin: m for **Meaning**, f for **Fun** etc.. We suffix with a star (*) variables bound to sequences. Syntactic forms such as **if** ... **then** ... **else** ... **endif** or **let** ... **in** ... are part of the metalanguage. For sum domains, if ϵ belongs to $\mathbf{S} = \mathbf{D}_1 + \ldots + \mathbf{D}_n$ then **sumcase** ϵ ... **endsumcase** dispatches on ϵ to the particular subdomain of the union of which ϵ is a member. For example, $(\epsilon)^{\mathrm{S}}\!\downarrow_{\mathrm{D}}$ is the projection of ϵ from **S** to **D** and $(\epsilon)_{\mathrm{D}}\!\uparrow^{\mathrm{S}}$ is the reciprocal injection from **D** to **S**.

Conditional expressions (`if`) and quotations (`quote`) are omitted since they are not relevant to our current discussion. Again, to simplify discussion, parameter evaluation is defined as left-to-right. More interesting are the three additional parameters in the semantic functions: ν_m, μ and δ. They represent the current module name, the module environment and the loaded environment, respectively. Their precise definition will appear later. The name of the current module is not really necessary but will be needed when discussing `eval` semantics later on. The module environment maps identifiers to modules while the loaded environment records which modules are loaded and their associated top-level environment. μ and δ respectively represent the static and dynamic components of modules. Note that μ is constant throughout these equations. A compiled application closes the module environment since it incorporates the compiled code of the modules it knows at compile- or link-time[8]. The other parameter (δ) is single-threaded and follows the store (σ) route.

The denotational equations for this Scheme-Lisp-like language are:

$\mathcal{E}$: **Form** $\rightarrow$ **Meaning**

$$\mathcal{E}[\![\nu]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m = \kappa_m(\sigma_m(\rho_m(\nu)), \sigma_m, \delta_m)$$

$$\mathcal{E}[\![\texttt{(setq }\nu\ \pi\texttt{)}]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m = \mathcal{E}[\![\pi]\!](\nu_m, \mu_m, \rho_m, \sigma_m, \delta_m,\ \lambda\ \varepsilon_c\sigma_c\delta_c\ .\ \kappa_m(\varepsilon_c, \sigma_c[\rho_m(\nu) \rightarrow \varepsilon_c], \delta_c))$$

```
$\mathcal{E}[\![\texttt{(lambda }\nu^*\ \pi\texttt{)}]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m =$
 let $\phi = \lambda\ \varepsilon_f^*\sigma_f\delta_f\kappa_f$ . if $\sharp\varepsilon_f^* = \sharp\nu^*$                    — check arity
                   then let $\alpha^* = new-locations(\sigma_f, \sharp\nu^*)$
                        in $\mathcal{E}[\![\pi]\!](\nu_m, \mu_m, \rho_m[\nu^* \rightarrow \alpha^*], \sigma_f[\alpha^* \rightarrow \varepsilon_f^*], \delta_f, \kappa_f)$
                   else $wrong$(“Wrong Number of Arguments”)
                   endif
 in $\kappa_m((\phi)_{\mathrm{Fun}}\!\uparrow^{\mathrm{Value}}, \sigma_m, \delta_m)$
```

$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m =$
$\mathcal{E}[\![\pi]\!](\nu_m, \mu_m, \rho_m, \sigma_m, \delta_m,$

[7] We will see that the introduction of macros able to create syntactically recursive procedures will pose some problems concerning the equivalence between interpreter and compiler.

[8] That allows partial linking at module definition time i.e.to collect modules definitions into a single logical one with all (now) internal references *hardwired*.

```
    $\lambda\ \varepsilon_c\sigma_c\delta_c$ . sumcase $\varepsilon_c$                    — check functionality
            Fun : $(\mathcal{E}^*[\![\pi^*]\!])(\nu_m, \mu_m, \rho_m, \sigma_c, \delta_c,$
                        $\lambda\ \varepsilon_{nc}\sigma_{nc}\delta_{nc}\ .\ ((\varepsilon_c)^{\mathbf{Value}}\downarrow_{\mathbf{Fun}}(\varepsilon_{nc}, \sigma_{nc}, \delta_{nc}, \kappa_m))$
            otherwise : wrong(“Not a function”)
          endsumcase )
```

$\mathcal{E}^* : \mathbf{Form}^* \to \mathbf{Meaning}$ — *ako* `evlis`

$\mathcal{E}^*[\![]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m = \kappa_m(<>, \sigma_m, \delta_m)$

```
$\mathcal{E}^*[\![\ \pi\ \pi^*]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m=$
 $\mathcal{E}[\![\pi]\!](\nu_m, \mu_m, \rho_m, \sigma_m, \delta_m,$
     $\lambda\ \varepsilon_{c1}\sigma_{c1}\delta_{c1}\ .\ \mathcal{E}^*[\![\pi^*]\!](\nu_m, \mu_m, \rho_m, \sigma_{c1}, \delta_{c1},\ \lambda\ \varepsilon_{c2}\sigma_{c2}\delta_{c2}\ .\ \kappa_m(<\varepsilon_{c1}>\S\varepsilon_{c2}, \sigma_{c2}, \delta_{c2})))$
```

This semantics is standard and should not pose problems. There is only one lexical environment (ρ) for variables. There is no point in checking if a variable is in ρ since the semantics of `defmodule` will ensure that any free variable is associated to a top-level location. Note that for any piece of program, the current ρ always extends the top-level environment of the module wherein that piece of code appears. This matter will be clarified when we get to the semantic definition of `defmodule`.

A module is a 3-tuple containing the list of the exported identifiers, the quasifunction that will load the module (it is a *quasifunction* since it behaves like a function except that it does not take arguments) and the quasifunction which is able to reinitialize the module — that is, reevaluate its body in its own top-level environment. **LoadEnv** maps module-names to their top-level environments and also keeps track of whether a module is loaded or not. These new domains are:

$\delta \in$	**LoadEnv**	=	$\mathbf{Id} \to \mathbf{LexEnv} \cup \{\textit{not-loaded}\}$
	QuasiFun	=	$\mathbf{Store} \times \mathbf{LoadEnv} \times \mathbf{Cont} \to \mathbf{Answer}$
	Module	=	$\mathbf{Id}^\star \times \mathbf{QuasiFun} \times \mathbf{QuasiFun}$
$\mu \in$	**ModEnv**	=	$\mathbf{Id} \to \mathbf{Module} \cup \{\textit{modenv-module-not-found}\}$

The description of `defmodule` was given informally earlier in this section. We now give a denotational description. Some utility semantic functions are used: $\mathcal{CI}$ is a predicate checking importations, $\mathcal{LV}$ extracts local names — the name under which it has been imported — from importations, $\mathcal{FV}$ extracts free variables from the body of the module ($\mathcal{FV}$ is not given here), $\mathcal{CE}$ is a predicate checking exportations, $\mathcal{L}$ loads (if not yet loaded) imported modules, $\mathcal{R}$ retrieves actual locations of imported modules and *augment−locations* adds fresh locations needed for free variables in the proper part of the top-level environment. The result of a `defmodule` pseudo special form is an extended **ModEnv**. The $\mathcal{M}$ valuation function gives a meaning to modules. $\mathcal{M}$ takes and returns an environment mapping names to modules.

$\mathcal{M} : \mathbf{Form} \to \mathbf{ModEnv} \to \mathbf{ModEnv} \cup \{\textit{incorrect-module}\}$

```
$\mathcal{M}[\![$(defmodule $\nu\ \iota^*\ \nu_e^*\ \pi$)$]\!]\mu_m=$
 if $\mu_m(\nu) = \textit{modenv-module-not-found}$
 then  if $(\mathcal{CI}[\![\iota^*]\!])(\mu_m)$                    — check importations
       then  let $\nu_l^* = \mathcal{LV}[\![\iota^*]\!]$               — extract local variable names from importations
             and $\nu_f^* = \mathcal{FV}[\![\pi]\!]$                 — extract free variables from the body
             in  if $\mathcal{CE}[\![\nu_e^*]\!]\nu_f^*\nu_l^*$                — check exportations
                 then $\mu_m[\nu \to \textit{make-module}(\nu, \iota^*, \nu_e^*, \nu_f^*, \nu_l^*, \pi, \mu_m)]$
                 else incorrect-module
                 endif
       else incorrect-module
       endif
 else incorrect-module
 endif
```

```
$\textit{make-module}(\nu, \iota^*, \nu_e^*, \nu_f^*, \nu_l^*, \pi, \mu_m) =$
$<\nu_e^*,$                                     — list of exported variable names
  $\lambda\ \sigma_i\delta_i\kappa_i\ .$                               — module initializer
```

```
$(\mathcal{L}[\![\iota^*]\!])(\mu_m, \sigma_i, \delta_i,$                                  — load unloaded imported modules
        $\lambda\ \varepsilon_{c1}\sigma_{c1}\delta_{c1}\ .$
          let $\rho = (\mathcal{R}[\![\iota^*]\!])(\delta_{c1})$                           — retrieve imported bindings
          in $augment-locations(\nu_f^*, \rho, \sigma_{c1},$                             — extend with proper bindings
            $\lambda\ \rho\sigma\ .\ (\mathcal{E}[\![\pi]\!])(\nu, \mu_m, \rho, \sigma, \delta_{c1},$                   — evaluate the body
                       $\lambda\ \varepsilon_{c2}\sigma_{c2}\delta_{c2}\ .\ \kappa_i(\varepsilon_{c2}, \sigma_{c2}, \delta_{c2}[\nu \rightarrow \rho])))),$
$\lambda\ \sigma_l\delta_l\kappa_l\ .\ (\mathcal{E}[\![\pi]\!])(\nu, \mu_m, \delta_l(\nu), \sigma_l, \delta_l, \kappa_l) >$                — module reinitializer
```

The `start-module` pseudo special form has the denotational description given below. Note that the module is first installed (that is, its imported modules are loaded and its body is evaluated) and then the specified entry point is sought in the resulting top-level environment, checked to be a function and called with the arguments provided by the operating system or whatever starts the module. In fact, the OS-arguments (ϵ_{os}) are then *incorporate*-d in the current store[9] and then control transfers to the entry point. The final continuation κ_{final} is implementation dependent, although it probably exits the application and returns to the caller whatever that may be. Note also that initial (i.e.empty) store σ_{init} and loaded environment δ_{init} are supplied when the module is started.

```
$\mathcal{M}[\![$(start-module $\nu_m\ \nu_\phi\ \epsilon_{os}$)$]\!]\mu_m =$
 if $\mu_m(\nu_m) = modenv\text{-}module\text{-}not\text{-}found$
 then $incorrect\text{-}module$
 else $(\mu_m(\nu_m) \downarrow 1)(\sigma_{init}, \delta_{init}$
          , $\lambda\ \varepsilon_{c1}\sigma_{c1}\delta_{c1}\ .$ sumcase $\sigma_{c1}((\delta_{c1}(\nu_m))(\nu_\phi))$
                       Fun : $(incorporate(\epsilon_{os}))(\sigma_{c1},\ \lambda\ \varepsilon_{c2}\sigma_{c2}\ .$
                                    $(\sigma_{c2}((\delta_{c1}(\nu_m))(\nu_\phi)))^{\text{Value}}\downarrow_{\text{Fun}}$
                                      $(<\varepsilon_{c2}>, \sigma_{c2}, \delta_{c2}, \kappa_{final}))$
                       otherwise : $wrong($"not a functional entry point"$)$
                      endsumcase )
 endif
```

The `load-module` special form (re-)initializes a module. Its denotational description is given below. It simply reevaluates the body form in the module's top-level environment. The top-level environment referred to is only created once, when the module is first loaded. An interesting observation is that although it might seems pointless to load a module unrelated to an application, because that module might share locations with the application, loading it might modify the contents of these locations. Consequently, we do not require the module ν to be in δ_m, but permit arbitrary loading of any known module[10].

```
$\mathcal{E}[\![$(load-module $\nu$)$]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m =$
 if $\mu_m(\nu) = modenv\text{-}module\text{-}not\text{-}found$
 then $wrong($"no such module"$)$
 else if $\delta_m(\nu) = not\text{-}loaded$
      then $(\mu_m(\nu) \downarrow 1)(\sigma_m, \delta_m, \kappa_m)$                 — initialize the module
      else $(\mu_m(\nu) \downarrow 2)(\sigma_m, \delta_m, \kappa_m)$                 — reinitialize the module
      endif
 endif
```

The supporting semantic functions for the module semantics are given in appendix A.

2.7 Practical Variants

>> The `defmodule` facility we have described has a rather crude syntax. It is not hard to devise something much more convenient — importing each binding one-by-one is tedious, much better would be the means to import all the bindings, or a selection of the bindings of a module in one go. Specifying both the exported-name and the imported-name, although useful to circumvent occasional name-clashes, is tiresome — in general, it suffices to use the same name as that with which the binding was exported.

[9] The incorporation is what does the usual `read` function.
[10] This capability will prove to be useful for macros.

>> Another extension to importation is to have local importations. For example, a syntax such as:
`(access` *module-name exported-name*`)`

could designate a precise and exported location from a module named *module-name*. This syntax could be used in the context of a variable reference or variable assignment. This local importation facility also known as a *qualified name* will prove to be useful when discussing macros to avoid capture problems. An equivalent but not unusual syntax makes use of a colon between the module name and the name of the variable such as *module-name*:*exported-name*. Note that this syntax is given a meaning at module definition time and not at parse-time as in Common Lisp.

>> Exportations have also suffered from our desire for simplicity and are specified as part of the syntax of a module in symmetry with imports. It is not necessary to be so strict. We hereby propose an extended syntax, more convenient than the previous spartan one, to deal with importations and exportations. We wanted this syntax to be highly readable and to correspond to a simple mental model. It is based on a notion of sequence of names standing for bindings and is nearly symmetric as for import or export. An importation or exportation is a *directive* with one of the following syntax:

```
directive  ::=  (expose module-name)
             |  (only (names...) directive)
             |  (except (names...) directive)
             |  (rename ((new-name old-name)...) directive)
             |  (union directive...)
             |  (mutable directive)
             |  (immutable directive)
```

- `expose` produces a sequence of binding names which correspond to the exported environment of *module-name*. Recall that no duplicated names can exist in an exported environment.
- A sequence of names can be restricted by filtering *only* names from a given set or *except* names from a given set with respective help from `only` or `except` directives.
- Bindings can be renamed thanks to `rename`. Note that it is legal to simultaneously exchange names such as in `(rename ((a b)(b a))` *directive*`)`. Nowhere can a sequence contain an ambiguity i.e.a name-clash so creating duplicates will raise an error.
- It is of course possible to merge directives with `union`. Once again the resulting sequence is checked to not contain duplicates.
- On the practical side, we add two specific filters controlling the mutability of bindings. They convert a sequence of names into the same sequence of names but with the according mutability. Of course it is a syntactic error to try to turn a previously immutable binding into a mutable one. This extension shows that other directives can also be added[11].

The model is nearly symmetric. The only discrepency is that we allow module M to write `(expose` M`)` to export all its toplevel bindings whilst we do not allow to import oneself.

Let us give an example of an hypothetic module providing the usual library of Lisp functions: `(defmodule Library`

```
(expose assembly-defined-procedures)      ;;importations
(union                                    ;;exportations
      (rename ((call/cc call-with-current-continuation))
             (expose assembly-defined-procedures) )
      (expose Library) )
(setq cadr (lambda (e) (car (cdr e))))
... )
```

All bindings of `assembly-defined-procedures` (including for instance `car`, `cdr`, `call-with-current-continuation` ...) are reexported as they are thanks to `(expose Library)`. New functions defined in the module itself such as `cadr` are also exported in the same go. A short nickname (`call/cc`) is also provided. Note that `(eq call/cc call-with-current-continuation)` is always true since the two names refer to the same location whatever content it has !

[11] We will see an example in section 3.12.

>> In addition to the lexical environment, Lisp also have a dynamic variable binding environment. It may be desireable that the operation of loading a module be affected by the contents of this dynamic environment. For example: `(dynamic-let ((*pi-precision* 18))`
`(load-module trigonometry))`
will set up the `pi` constant — a top-level variable in the `trigonometry` module — to an accuracy of 18 digits.

>> Having `load-module` as a special form does not prevent us to have a *function* which does the same thing: say `(dynamic-load-module '`*module-name*`)`. The advantage of the `load-module` special form is that since the parameter is not evaluated it is possible for a program to determine statically the modules required to build a stand-alone application. Should the capability for dynamic loading of modules be present, the run-time system must be extended to cope with this. If the dynamic loading capability is kept in the `dynamic-loading` module, then in the absence of `eval` or `symbol-function`, it can be determined statically if the facility is required in an application. These properties are crucial if one wants to build small applications easily.

>> Modules can be mapped onto files provided that the quasi special form `defmodule` is able to compile a file as a module. The file may contain two initial expressions respectively corresponding to the importations and exportations followed by a sequence of expressions: its body. We do not address in this paper, problems related to macrocharacters. Outputs of the module definition can be an interface file describing imported modules and exported locations as well as an object code file, a "`.o`" file in the UNIXterminology.

2.8 Conclusion on Modules

We have been defining an essential module facility which fulfills different goals. First, modules allow information hiding via import/export clauses, second, the concept of application, module definition and module loading are now clear. Several benefits arise which mainly are — separate compilation, — partial linking, — and selective linking. This module facility while keeping with the Lisp spirit clearly separates linguistic from environmental issues. With such a facility, some static analyzes such as "are locations initialized ?" or "does this variable need to be recomputed if the module is to be reloaded ?" can be performed since all uses of variables are known in the module and mutability of their possible exportations is also known. Note that such analyzes circumvents the fuzzy behaviour of `define` in Scheme as well as the notion of *integrable* procedures. To partly answer these questions tremendously improves the efficiency since a location which is guaranteed to be initialized can be read without having to check if it contains the "undefined" value. A location which is always bound to a value which does not need to be recomputed, i.e.a constant, can be propagated or inlined with benefits.

3 Macro Expansion

Much of the power of Lisp stems from macros, but while their normal use does not pose problems for users, their semantics is somewhat bizarre. Macros provide a way to extend the syntax of Lisp. Such extensions are specified by rewriting rules which computation is expressed in Lisp and which aspect mimics combinations. This funds the power of macros but blurs the linguistical borders between definition and evaluation both in time and space. Under the "macro problem" title many subproblems can be identified and were attacked. For instance Kohlbecker [Kohlbecker *et al.*, 1986], Bawden and Rees [Bawden & Rees, 1988] addressed the name capture subproblem while Dybvig, Friedman and Haynes [Dybvig *et al.*, 1988] addresses the code-walking subproblem with the Expansion Passing Style (EPS). Neither of them analyse the relation between macros and modules which was only recently addressed [Curtis, 1990].

The next section will present examples illustrating the different macro subproblems. Some of them are somewhat artificial since most of these problems are interconnected and to focus on one tends to neglect the others. Current solutions to these problems are also covered. The semantics of macros is finally presented in section 3.11.

If we were to shortly express the problems lying with macros, we would say "when, where, and how" ! Since macroexpansion is a Lisp computation, it has to be clearly defined with respect to which environment and continuation are performed macroexpansions, what is the lifetime of side-effects, etc.

3.1 The Status of Macros

A macro call may be viewed as a directive to the macroexpansion process to rewrite the directive itself into another text. A macro call mimics a function call in that it uses a similar syntax i.e.a non empty list. A keyword stands in the `car` of this list and is associated to an *expander*: a regular function that rewrites the text of the macro call i.e.computes another text to replace the original one.

The usual shape of a macro call is a list but it is also possible to have *macro symbols*: the rewriting rule is triggered by the occurrence of the macro symbol[12]. More generally the entire text of the module has to be walked in order to produce a fully expanded text which must be a regular form. This leaves room for general walkers performing non trivial tasks such as curryfication or introduction of call by need [Dybvig *et al.*, 1988]. This description does not constrain a macro to be a first class object nor it allows to `apply` macros, to retrieve them by their associated keyword ... We thus keep macros outside first class objects and let them deal only with syntax.

3.2 The Name Capture Problem

The result of an expression may refer to names that are free. They may thus be assigned a meaning depending on the exact place where they are inserted. Consider for instance:

```
(defmacro push (item place)
  `(setq ,place (cons ,item ,place)) )
```

`setq` and `cons` appear free in the expansion, they are susceptible to be captured[13] as in[14]:

```
(macrolet ((setq (name f
  (flet ((cons (x y) ...))
     ... (push α β) ... ) )
```

The result of the expansion although correct from the viewpoint of the implementor of the macro is likely to be erroneous since `setq` and `cons` have probably incompatible local meaning.

Some solutions were studied. Hygienic expansion [Kohlbecker *et al.*, 1986] proposes a scheme where potential name captures are predicted and implicitely alpha-converted avoiding the need to explicitely call `gensym`.

Another solution is provided by syntactic closure [Bawden & Rees, 1988]. Programs are written according to a precise syntax. Syntax may be locally modified and program excerpts may be closed in a given syntax. Some "holes" may be offered[15] within a closed expression in order to freely fill these holes with other closed expressions. For instance the expansion of the previous `push` macro has to be considered with respect to the syntax that was current where the `push` macro was defined. The `push` expansion contains two holes that will be filled with expressions closed in the syntax that will be current at the point where the `push` macro will be called.

A third solution is to explicitely "qualify" variables (i.e.specifying their original module whre they were defined) and write `(access standard cons)` instead of `cons`[16]. This provides only a partial solution since special form names can still be captured. This can be solved if we control the macroexpansion process with EPS for instance.

3.3 The Walking Algorithm

Some macros perform side-effects. For instance a `defclass` macro as provided by an object oriented layer, enriches the hierarchy tree (or dag) of classes. Side effects can only be mastered if one can control time or sequentiality i.e.the order along which are performed macroexpansions. Consider the following queer macro:

```
(defmacro foo (x y)
  (if (evenp (incf *foo-counter*)) x y) )
```

It is not hard to devise some variants fo the macroexpansion algorithm that takes `(foo (foo a b) (foo c d))` and returns `a`, `b`, `c` or `d` wether the algorithm is bottom-up or top-down, left to right or right to left[17] and given the initial parity of `*foo-counter*`.

Another queer macro emphasizes the relative timing of macroexpansion versus compilation. Consider for instance:

```
(defmacro memorize (&whole form)
   (push form *the-memorized-forms*)
   `(quote wait-a-little) )

(defmacro end-of-memorize ()
   (dolist (form *the-memorized-forms*)
     (setf (cadr form) (length *the-memorized-forms*)))
```

[12] `symbol-macrolet` is an example of this feature which had recently been added to Common Lisp.

[13] The syntax of the following `macrolet` is taken from Common Lisp.

[14] We neglect the possible problem that the `setq` special form may be implemented as a macro that can be shadowed by a local macro definition. Often special forms cannot be altered nor shadowed.

[15] Actually they are restricted to be only symbols.

[16] Of course, we suppose that the original `cons` function is defined in the `standard` module.

[17] Scheme does not imposes the order along which are computed the arguments of a combination. Macroexpansion may also take this view.

```
    (length *the-memorized-forms*))
```

The `memorize` macro is strange because it collects information from all the places it is used until `end-of-memorize` eventually modifies all the temporary expansions previously performed. Unless macroexpansion is done incrementally — separate from compilation — then this behaviour would work. However, it imposes the additional constraint that syntactic checks cannot be done before the macroexpansion process is finished. This, in turn, requires two walks of the text of the module but is clearer from the semantic point of view.

A solution proposed in [Curtis, 1990] is to leave the macrowalk algorithm completely undeterminate. While this undeterminacy fits well with the Scheme semantics of combination, it ignores the sequentiality of other special forms such as `if` or `progn`. Therefore apart embedding one's code in a macro which performs its won code-walking, mastering side-effects seems difficult. Although partially unknown the usual macrowalking algorithm seem to be depth-first and left to right: the second solution is to publish the exact macroexpansion algorithm and to let the user provide its own way of expanding expressions as done in EPS. This allows to extend the macrowalking algorithm by appropriate "methods" sparing the burden to write such a walker.

The whole macroexpansion process is related to syntax and must yield a pure expression that can be further transmitted to the evaluator or module definer.

3.4 Expansion at Will

It is sometimes useful to be able to macroexpand at will some expressions. This facility is usually offered by the mean of the `macroexpand` function. The `macroexpand` function takes a text, walks it according to the current syntax and returns a new text. If the syntax can be locally modified then different versions of `macroexpand` exist corresponding to the various syntaxes. This situation is recognized by EPS [Dybvig *et al.*, 1988] which allows the user to get the current `macroexpand` when performing a macroexpansion. An expander is thus a function that takes a text to macroexpand and the current macroexpand function (which in a way closes the current syntax). The expander returns a text which replaces the original one. This result is considered as definite i.e.is not walked any more. What is the signature of `macroexpand` ? Clearly it must take a text to macroexpand but since this text may contain embedded macro calls, it must provide the current `macroexpand` to these macro calls. So `macroexpand` has the signature: `(lambda (`*text macroexpand*`)` *text*`)`

We can take benefit of this facility and propose to rewrite the `push` expander as: `(lambda (text macroexpand)`

```
  (let ((item (macroexpand (cadr text) macroexpand))
        (place (macroexpand (caddr text) macroexpand)) )
    `(setq ,place ((access standard cons) ,item ,place)) ) )
```

The contract of an expander is to return a fully expanded text by recursively expanding, with the appropriate macroexpanders, the various subparts of the macro call. The result must be a pure text of the language and does not need to be reexpanded. This contrasts with the usual behaviour of macros (as in Common Lisp) where macroexpansion is performed again and again until the text becomes unexpandable i.e.an atom or a list which `car` is not a macro keyword. Control is then given back to the hidden walker which tries to expand subparts that may still contain macro calls.

The above described macroexpansion looks like syntactic closures except that texts are not closed in their syntax but macroexpanded according to the current syntax until becoming a regular expression.

3.5 Compiler Macros

Macros are sometimes used to perform some optimizations that the compiler may not discover by itself. Consider for instance the `acons` function: `(define acons (lambda (key val alist)`

```
                 (cons (cons key val) alist) ))
```

To avoid the cost of calling `acons`, `acons` may also be associated to a macro: `(defmacro acons (key val alist)`

```
  `((access standard cons) ((access standard cons) ,key ,val)
    ,alist ) )
```

Now consider the following excerpt: `(acons 'acons acons '())`

If we were to retain only one of the two definitions of `acons`, we would loose or the inlining of `acons` or the functional value of `acons`. Neither of both is acceptable according to current practices.

A first solution is to inform the compiler that the `acons` function can be inlined. This can be done by a `declare` or `proclaim` form as in Common Lispbut may also be inferred by a very clever compiler. The problem is that compilers do not guarantee to take this declaration into account. To thus define `acons` as a macro guarantees the inlining to be performed but it then seems difficult to extract the function definition from the macro definition and to provide where necessary the functional value of `acons`. The best way is to combine the two approaches: `acons` is defined as a function, an expander is derived from this function and whenever `acons` is imported, the expander for `acons` enriches the importing syntax. Now, depending on the context where `acons` occurs, functional calls to `acons` will be expanded while references to the `acons` variable will be left in place. We will see later how to enrich a syntax in such a way.

This solution mimics Common Lispcompiler-macros but has a clearer semantics since it separates the two involved effects: the semantical and the syntactic one.

3.6 The Language of Macros

What is the language in which are expressed macros ? For sure they are written in Lisp ! The question is not so naïve. Can the body of a macro use an extended syntax i.e.can it use macros itself and which ones ?

A macro is associated to a regular function: an expander. This function has to be defined in some module which clearly specifies the set of bindings that can be referred or altered but also the syntax of the expander i.e.what macros it can use. This view has a drawback in that the expander can only be used after the module had been loaded, therefore a macro cannot be used inside the module it is defined in[18].

There exists a predefined syntax described in section 2.6 roughly equivalent to the essential syntax of Scheme. The initial `macroexpand` knows how to code-walk an expression written according to this syntax. We propose that syntax can be modified thanks to: `(with-syntax (`*module-name exported-name*`)` *expression* `)`

The meaning of this syntactic form is:

- load, if needed, the *module-name* module,
- retrieve the value of the *exported-name* variable which must be exported from *module-name* and must have a functional value. This function will be known as a *syntax modifier.*
- apply the syntax modifier to the current syntax i.e.the current `macroexpand`,
- the result is the new `macroexpand` that will be used to macrowalk the *expression*.

The signature of a syntax modifier is then: `(lambda (`*macroexpand*`)` *macroexpand*`)`

This form allows to introduce new macros as in: `(define push-macro` ;; *This definition belongs to the* `standard-macros` *module.*

```
  (lambda (syntax)
    (extend syntax 'push
            (lambda (text macroexpand)
               (let  ((item (macroexpand (cadr text) macroexpand)
                      (place (macroexpand (caddr text) macroexpand) )
                  `(setq ,place ((access standard cons) ,item ,place) ) ) ) ) ) )
```
;;; *One can extend syntax thanks to* `extend`
```
(define extend
  (lambda (syntax keyword expander)
    (lambda (text macroexpand)
      (if (and (pair? text)
               (eq? (car text) keyword) )
          (expander text macroexpand)
          (syntax text macroexpand) ) ) ) )
```
;;; *and one can use the* `push` *macro as in*
```
 ... (with-syntax (standard-macros push-macro)
          ... (push Q P) ... ) ...
```

A more convenient facility can be introduced to lessen the burden of macro definition: the `defmacro` macro. If we suppose to already benefit of `defmacro`, it will be defined as: `(defmacro defmacro (name variables . body)`
```
  `(lambda (syntax)
     (extend syntax ',name
             (lambda (text macroexpand)
```

[18] We will nevertheless present later a solution to overcome this defect.

```
                    (macroexpand (apply (lambda ,variables . ,body)
                                     (cdr text) )
                              macroexpand ) ) ) ) )
```

The `with-syntax` syntactic form also allows to introduce locally specialized syntax. The following function `(lambda (syntax) standard-macroexpand)` is a syntax modifier that reestablishes a pure syntax without any macros. This feature allows to juggle with syntaxes like syntactic closures except that we do not introduce a new concept but turn texts into the lingua franca: the pure syntax with its limited set of special forms. `with-syntax` is not a special form, it is only a syntactical form (like `access`) i.e.a directive that will be obeyed during macroexpansion. Note that this capability to introduce a peculiar local syntax is not possible in a model (as in usual Lisps) where the result of a macroexpansion is reexpanded at its calling site.

The initial question is then answered: a macro is supported by a regular function, a syntax modifier, locally introduced by `with-syntax`. The syntax modifier obeys to the syntax of the module it belongs to.

3.7 Scope of Macros

In usual Lisp sytems, macros are defined thanks to `defmacro` or `macro` forms. These definitions have problems related to scope and availability. Besides these forms also exist a local macro definition form such as `macrolet` in COMMON LISPwhich does not have these problems. Consider the following definition: `(defmacro foo ...)`

Where is available the macro `foo` ? In the whole module containing `foo`, everywhere after the definition of `foo` or outside the module. A functional definition is available in the whole module since the toplevel environment provides an implicit `letrec`. If a macro is to have the same capability, then nothing can be known before the body of the module has been inspected to know which are the macros. But now what if macros are redefined ? And what if macros generate new macros ? Clearly this view suggests many problems.

Consider now that the macro is only available after being defined i.e.something like: `(progn (defmacro foo ...)`
α `)`

which may be considered as equivalent to: `(macrolet (foo ...)`
α `)`

This view generally imposes the concept for *toplevel forms* so that deeper macros can be seen as in:
```
(progn (progn (defmacro foo ...)
              α )
    β )
```

since most people probably want β to be aware of `foo`. The concept of toplevel-forms introduce a grammar which indicates which forms in a module can be considered as toplevel. If a toplevel form is a macro definition then it will be available for all following forms.

But two kinds of macros can be recognized wether one wants to export them or not. To assimilate macros to local ones as we did above denies the exportability of them.

Another problem lies with the language that can be used to define the macro. We said that it should be the language of the module so a macro can use the macros of the module as well as the variables and functions of the module and that is clearly bizarre since macroexpansion takes place before these variables and functions are defined. This leaves with a single solution, macros cannot be used in the module where they are defined and `with-syntax` is the only way to introduce them solving the problems of scope (the body of `with-syntax`) and availability (outside the module where the macro is defined).

3.8 Termination of Macroexpansion

If macros are functions that convert syntax to other syntax by an unrestricted computation then they cannot be guaranteed to terminate in all cases (though the user is suggested to try to). The default of that is that a program using macros cannot be denoted by standard denotational semantics since to fully expand the body of a module is a computation that cannot be bounded in time.

The denotation we gave for our language used two semantical functions $\mathcal{M}$ and $\mathcal{E}$ dealing with the modules and the rest of the language. The semantics of a module was given in 2.3. To deal with macros just imply to modify the step 2 as such:

2a the body of the module is converted into an S-expression ϵ_0. This value belongs to the **Value** domain and cannot be considered independently of the store σ_0 where it lies.

2b This S-expression is macroexpanded and, if it terminates, yields a value ϵ_D in a store σ_D.

2c This value ϵ_D is converted into an element of the **Form** domain: π. Syntactic checks are made when translating ϵ_D into π.

2d the rest of the module definition is then resumed as explained in the original step 2 of section 2.3.

Step 2c may be viewed as if we were freezing the store σ_D i.e.the body of the module is immutable and cannot be altered by further use of physical modifiers such as `rplaca` or `rplacd`. Rather than converting a value into a form we can adapt the semantical equations defining $\mathcal{E}$ to directly deal with values (see figure 1).

$$\begin{tikzcd}
\text{Text} \arrow[r] & \mathbf{Value}\times\mathbf{Store}\ni(\epsilon_0,\sigma_0) \arrow[d] \\
\mathbf{Form}\ni\pi \arrow[d, "\mathcal{E}"'] & \mathbf{Value}\times\mathbf{Store}\ni(\epsilon_D,\sigma_D) \arrow[l] \arrow[dl, "\mathcal{E}'"] \\
\text{Denotation} &
\end{tikzcd}$$

Figure 1: The *denotation* of macros

Let us give a flavor of this new function $\mathcal{E}'$ which now mimics a classical interpreter:

$\mathcal{E}'(\lceil\texttt{(setq}\ \nu\ \pi\texttt{)}\rceil, \sigma_D)\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m =$
$\quad\mathcal{E}'(\lceil\pi\rceil, \sigma_D)(\nu_m, \mu_m, \rho_m, \sigma_m, \delta_m,\ \lambda\ \varepsilon_c\sigma_c\delta_c\ .\ \kappa_m(\varepsilon_c, \sigma_c[\rho_m(\lceil\nu\rceil) \to \varepsilon_c], \delta_c))$
$\quad$**where** $\lceil\nu\rceil = \sigma_D(\sigma_D((\lceil\texttt{(setq}\ \nu\ \pi\texttt{)}\rceil)^{\mathbf{Value}}\downarrow_{\mathbf{Pair}}\downarrow 2)^{\mathbf{Value}}\downarrow_{\mathbf{Pair}}\downarrow 1)$ — *ako* `cadr` *!*
$\qquad\quad \lceil\pi\rceil = \sigma_D(\sigma_D(\sigma_D((\lceil\texttt{(setq}\ \nu\ \pi\texttt{)}\rceil)^{\mathbf{Value}}\downarrow_{\mathbf{Pair}}\downarrow 2)^{\mathbf{Value}}\downarrow_{\mathbf{Pair}}\downarrow 2)^{\mathbf{Value}}\downarrow_{\mathbf{Pair}}\downarrow 1)$ — *ako* `caddr` *!*

σ_D is the definition store, σ_m is the store before evaluating π and σ_c the final store. There is no connection between σ_D and σ_m: the first one was frozen at module definition time while the second is the run-time store. Therefore no modification of σ_m can affect the meaning of programs within σ_D. We use $\lceil$ and $\rceil$ to delimit quasi-syntactic elements. They do not have any meaning, they only serve to delimit elements of **Value** that are considered as program text.

The other rules for $\mathcal{E}'$ follow a similar pattern.

3.9 Syntactically Recursive Programs

The result of a macroexpansion is a value. This value may greatly differ from an usual form: it can be a DAG or even contains cycles. Consider for instance the two examples:

```
(defmacro snoopy (&rest forms)
  `(progn ,@(mapcan #'(lambda (form) `(,form (printf ".")))
                    forms)))
(defmacro while (&whole form cond . body)
  (labels ((displace (old new)
             (setf (car old) (car new))
             (setf (cdr old) (cdr new))
             old))
    (displace form `(when cond (progn . ,body) ,form)))
```

The `snoopy` macro when invoked will return new forms that will probably share the representation of the subexpression `(printf ".")` i.e.DAGS. That will cause no harm. The other macro is more subtle since it generates cyclic forms in fact syntactically recursive forms. Most compilers will be defeated in trying to compile such forms. A compiler might memoize expressions to compile and convert syntactic fix-points to semantic fix-points when compiling the same expression in congruent lexical contexts, but this would be unusual. For example: `(while` α β`)` may be considered as the fix-point of the syntactic function from **Form** to **Form**:

$$\lambda\pi.[\![\texttt{(when}\ \alpha\ \texttt{(progn}\ \beta\texttt{)}\ \pi\texttt{)}]\!]$$

and may be converted to the fix-point of `(lambda (`Φ`) (when` α `(progn` β`) (`Φ`)))` that may be compiled as: `(letrec ((`Φ `(lambda () (when` α `(progn` β`) (`Φ`)))))`

```
    (Φ))
```

The validity of this particular example is easy to see, but it is also a general result. An exposition of the argument for the pure λ-calculus may be found in [Stoy, 1977, page 182]. The trick is to consider the valuation function $\mathcal{E}$ to be defined on the domain **Form** which besides being a syntactic domain may be defined as:

$$\begin{array}{rlll} \textbf{Form} = & \textbf{Id} & & \text{variable references} \\ + & \textbf{Id} \times \textbf{Form} & & \text{assignment} \\ + & \textbf{Id}^* \times \textbf{Form} & & \text{abstraction} \\ + & \textbf{Form} \times \textbf{Form}^* & & \text{combination} \end{array}$$

By this domain definition, syntactically recursive programs are now elements of **Form**. The equations for $\mathcal{E}$ define a functional $\lambda\mathcal{E}.\ominus(\mathcal{E})$ which is continuous: $\mathcal{E}$ exists and is its least fix-point. We can therefore denote a constructed value — an element of **Value** — with an element of **Meaning**. This result should not be surprising to Lisp programmers since a computed value can be processed by an interpreter even if it is syntactically recursive and denotational semantics may be (restrictively) viewed as the art of writing language interpreters in a very sparse language: the λ-calculus.

We have not considered the ill-formed forms, which are not members of the **Form** domain, such as this:

```
(let ((form (list 'setq x)))
  (setf (cadr form) form) )
→ (setq (setq (setq ...)))
```

Note also that since a value in **Value** is made of a finite number of dotted pairs and that only a finite number of syntactic forms exist then it is decidable to check the syntax of elements of **Value**.

The current solution is to ignore such cyclic forms i.e.the compiler loops and the user has to kill its compilation. Since no good programs are known to be only expressible by means of syntactically recursive programs, it seems luxurious to enable a compiler to process these forms. Note that interpreters also ignore cyclic forms since this is not a problem for them. Thus if one wants to have a perfect identity between a compiler and an interpreter, one has to deal with syntactically recursive forms which imposes an heavy burden on the compiler.

3.10 Mixing Macroexpansion Time and Run Time

Another problem with macros is that they are needed at module definition time but must not be needed anymore at run-time. The trouble does not lay with `with-syntax` since its semantics is clear and only restricted to the module definition time. The result of the macroexpansion, say (ϵ_D, σ_D), must not need anything related to the resources that were acted during macroexpansion. If analysing the special forms that may appear in the fully expanded body of the module, one can see that only one possible connection exists: the quotation. The quotation introduces a immediate datum: a S-expression. If this datum is a dotted pair, it has to be considered in its associated store and that connects the run-time store with the module definition-time store.

Several solutions exist to break this connection. A first one is to consider the definition store as immutable and to arrange the datum to be constructed in such a way that it cannot be mutated: that introduces new types such as immutable dotted pairs. A second solution is to consider that one can only quote immutable data such as numbers or symbols. Therefore a dotted pair has to be explicitely rebuilt at run-time if one wants such a constant. In such a world, to write `'(`α`.  `β`)` is equivalent to `` `(,' ``α`.  ,'`β`)`. We retain this solution which in fact corresponds to the reality of module compilation where constants appearing in a module have to be rebuilt into the store where the module is loaded.

The final definition for the quotation is therefore:

```
$\mathcal{E}'(\lceil\texttt{(quote }\epsilon_D\texttt{)}\rceil, \sigma_D)\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m =$
  sumcase $\epsilon_D$
   Id : $\kappa_m(\epsilon_D, \sigma_m, \delta_m)$
   Number : $\kappa_m(\epsilon_D, \sigma_m, \delta_m)$
   otherwise : $wrong(\text{``}\texttt{Incorrect quotation}\text{''})$
 endsumcase
  where $\epsilon_D = \sigma_D(\sigma_D((\lceil\texttt{(quote }\epsilon_D\texttt{)}\rceil)^{\textbf{Value}}\downarrow_{\textbf{Pair}}\downarrow 2)^{\textbf{Value}}\downarrow_{\textbf{Pair}}\downarrow 1)$— *ako* cadr !
```

3.11 Formal Semantics of Macros

The solution we propose along these sections is to insert macroexpansion during module definition. The macroexpansion is performed by an initial macroexpander which aim is to find `with-syntax` syntactical forms and to perform the needed walking according to these local syntaxes. The result is a neat expression only using the standard special forms. The denotation of **`defmodule`** thus becomes:

```
$\mathcal{M}[\![$(defmodule $\nu\ \iota^*\ \nu_e^*\ \pi$)$]\!]\mu_m =$
 if $\mu_m(\nu) = $ modenv-module-not-found
 then if $(\mathcal{CI}[\![\iota^*]\!])(\mu_m)$                          — check importations
        then $(incorporate[\![\pi]\!])(\sigma_{init},$
                                 $\lambda\epsilon_i\sigma_i.$
                                   $macroexpand(<\epsilon_i, macroexpand>,$
                                                $\sigma_i,$
                                                $\delta_{init},$
                                                $\lambda\lceil\pi\rceil\sigma_D\delta_D.$
        let $\nu_l^* = \mathcal{LV}[\![\iota^*]\!]$                    — extract local variable names from importations
        and $\nu_f^* = \mathcal{FV}'(\lceil\pi\rceil, \sigma_D)$            — extract free variables from the body
        in if $\mathcal{CE}[\![\nu_e^*]\!]\nu_f^*\nu_l^*$                       — check exportations
           then $\mu_m[\nu \rightarrow make\text{-}module'(\nu, \iota^*, \nu_e^*, \nu_f^*, \nu_l^*, \lceil\pi\rceil, \mu_m)]$
           else incorrect-module
           endif
        else incorrect-module
        endif
 else incorrect-module
 endif
```

```
$make\text{-}module'(\nu, \iota^*, \nu_e^*, \nu_f^*, \nu_l^*, \lceil\pi\rceil, \mu_m) =$
$<\nu_e^*,$                                       — list of exported variable names
  $\lambda\ \sigma_i\delta_i\kappa_i\ .$                                  — module initializer
   $(\mathcal{L}[\![\iota^*]\!])(\mu_m, \sigma_i, \delta_i,$                          — load unloaded imported modules
          $\lambda\ \varepsilon_{c1}\sigma_{c1}\delta_{c1}\ .$
           let $\rho = (\mathcal{R}[\![\iota^*]\!])(\delta_{c1})$                       — retrieve imported bindings
           in $augment - locations(\nu_f^*, \rho, \sigma_{c1},$               — extend with proper bindings
              $\lambda\ \rho\sigma\ .\ (\mathcal{E}'(\lceil\pi\rceil, \sigma_D)(\nu, \mu_m, \rho, \sigma, \delta_{c1},$        — evaluate the body
                              $\lambda\ \varepsilon_{c2}\sigma_{c2}\delta_{c2}\ .\ \kappa_i(\varepsilon_{c2}, \sigma_{c2}, \delta_{c2}[\nu \rightarrow \rho]))))),$
  $\lambda\ \sigma_l\delta_l\kappa_l\ .\ (\mathcal{E}'(\lceil\pi\rceil, \sigma_D)(\nu, \mu_m, \delta_l(\nu), \sigma_l, \delta_l, \kappa_l) >$   —   module reinitializer
```

Macroexpansion takes place right after checking imports and computes the expanded body of the module — an element of **Value**. The semantic function *incorporate* (as used in the **`start-module`** semantics) takes the text $[\![\pi]\!]$ and converts it into a value ϵ_i with a store σ_i, where it is *macroexpand*-ed into $\lceil\pi\rceil$ in σ_D. The semantic function *macroexpand* is an element of **Fun** and as such its application is not guaranteed to terminate. We denote modules in the obvious way, that is, we associate a λ-term with the text of a module, although we do not denote forms but instead provide an interpreter $\mathcal{E}'$ for elements of **Value**. Since the expression to interpret is closed in σ_D and cannot be altered, it can be compiled — translated[Kelsey & Hudak, 1989] — into a more efficient code: $\mathcal{E}'$ is simply partially evaluated (or constant folded) with respect to $\lceil\pi\rceil$ and σ_D. Note also that self modifying programs are not possible in that model nor are quotations (see [Queinnec 90] for such a model).

3.12 Variants

On the practical side, it is often important to export from a module functions and macros altogether. Of course they do not have a similar status since a macro cannot be applied since it is merely syntax but the user should not be too much aware of the nature of the exportations. The main reason is that macros are used inside **`with-syntax`** while functions or variables are imported in the importation clause. We thus propose a new importation (and symmetrically also, exportation) directive:

directive ::= `(syntax` *directive*`)`

The meaning of the **`syntax`** directive is to declare that the locations produced by the inner *directive* contain macros i.e.syntax modifiers. When all importations are collected, syntactical ones are extracted and composed with the initial macroexpander as if they were put in a serie of implicit **`with-syntax`** embedding the whole body of the module. This body itself is considered as embedded in an implicit **`progn`**. In other words the two following modules are equivalent: `(defmodule M`   `(defmodule M`

```
(syntax (only (push-macro)                  (union)
              (expose standard-macros) ))   (expose M)
(expose M)                                    (with-syntax (standard-macros push-macro)
  ... )                                           ... ) )
```

The imported bindings used for syntax are of course removed from the importations since they must not be kept at run-time. In the previous example, module `M` does not require module `standard-macros` after macroexpansion.

Since exportations and importations are symmetric, one can also export a function as a syntax modifier i.e.a macro. One can then impose to view a function as a macro outside. For instance: `(defmodule standard-macros`

```
(expose Library)
(syntax (expose standard-macros))
  (defmacro push (item place)
    ... )
  ... )
```

In the `standard-macros` module, syntax modifiers are defined and exported as syntax. The user has therefore no means to retrieve the expander of `push`, she can only use `push` as a syntactical tour-de-main.

3.13 Conclusions on Macros

We longly analyse various aspects of usual macros and provide much examples of their power and dysfunctionment. We generalize macros to be special code-walkers converting forms written with a local syntax into expressions of the pure language. An unique feature is provided to introduce local syntax: `with-syntax`. Macros do not exist per se but appear as syntax modifiers i.e.functions acting on syntax.

This approach has a great benefit since it clearly separates module-definition-time where macros are expanded and run-time from which macros are excluded. The cost is on the semantical side since we abandon denoting syntactical programs in favor of denoting values. We reestablish an interpreted semantics and underline the difficulty of ignoring the processor (compiler or interpreter) when processing syntactically recursive forms. Paradoxically the gain is semantic since now we gain a macro model where we know what happens.

The only restriction is that we cannot use macros inside the module where they are defined. We will see in the next section some ways to alleviate this restriction.

By the way the antique explanation of macros was “functions that do not evaluate their arguments but doubly evaluate their result”. Our description makes clear that the two involved evaluations belongs to different modules (the first one is performed in the module where the macro is defined while the other is performed in the module where the macro is called) and therefore are unrelated.

4 Dynamic Evaluation

$\mathcal{E}'$ is the explicit Lisp interpreter. It seems easy then to offer an `eval` facility. `eval` permits the dynamic synthesis of programs and thus demands the presence of at least an evaluator in applications which use `eval`. `eval` is one of the most typical features of Lisp but untamed it offers many nuisance. It is our aim to define a kind of domesticate `eval` that must not break the module system nor the macro feature. Moreover its cost must only be supported by its users. We will see in this section after a short analysis of `eval` features, the semantics of `eval` and some interesting uses of it.

`eval` allows to dynamically synthesize new code and to evaluate it. `eval` can be provided either as a function or as a special form.

If `eval` is a function and if we do not change the previous denotation rules, `eval` cannot capture the current lexical environment. The argument of `eval` will then be evaluated in the environment where `eval` was defined — that is, in the top-level environment of the module to which it belongs. The expression being evaluated can only use the resources that are visible from the `eval` module, which excludes any user resources. This fact makes such an `eval` practically useless.

If `eval` is a special form, it could be defined to capture some environment. The first solution is to offer what might be called `eval-in-current-context`. The parameter of this kind of `eval` will be evaluated in the environment of the call to `eval`. This solution clearly requires that all intermediate compilation structures must be retained for each place where an `eval` form occurs. For instance: `((lambda (x y) (eval x))`
`'y 'foo )` $\rightarrow$ `foo`

In this example the name of the visible bindings `x` and `y` must be kept in order to transparently evaluate any expressions. `eval-in-current-context` may be viewed as a sort of reciprocal of the quotation since `(eval (quote` α`))` is, in any context, totally equivalent to α.

The latter sort of `eval` is very costly and many Lisp systems prefer to offer a different `eval` which only capture the top-level environment of the module in which the `eval` form appears. A more appropriate name for that `eval` is `eval-in-current-module` or `eval/cm` for short. The compiler must still be present in any application which uses `eval` but only the symbol table of the current module must be retained. Moreover this `eval` feature recovers a functional definition and does not need anymore to be a special form. We therefore propose to keep this facility and to define it more precisely.

4.1 Formal Semantics for `eval/cm`

The formal semantics of `eval/cm` is simple:

$$\begin{array}{l}
\mathcal{E}'(\lceil\texttt{(eval/cm }\pi\texttt{)}\rceil, \sigma_D)\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m = \\
\quad \mathcal{E}'(\lceil\pi\rceil, \sigma_D)(\nu_m, \mu_m, \rho_m, \sigma_m, \delta_m,\ \lambda\ \varepsilon_c\sigma_c\delta_c\ .\ \mathcal{E}'(\lceil\varepsilon_c\rceil, \sigma_c)(\nu_m, \mu_m, \delta_m(\nu_m), \sigma_c, \delta_c, \kappa_m) \\
\quad \textbf{where } \lceil\pi\rceil = \sigma_D(\sigma_D((\lceil\texttt{(eval/cm }\pi\texttt{)}\rceil)^{\textbf{Value}}\downarrow_{\textbf{Pair}}\downarrow 2)^{\textbf{Value}}\downarrow_{\textbf{Pair}}\downarrow 1) \text{— } \textit{ako } \texttt{cadr} \textit{ !}
\end{array}$$

Note the use of the toplevel environment of the current module $\delta_m(\nu_m)$ when evaluating the argument of `eval/cm` $\lceil\varepsilon_c\rceil$.

It is also necessary to revise the formal semantics of reference and assignment since it is possible now to dynamically synthesize references or assignments on never seen before variables. These variables will be added to the toplevel of the module but will not be modified if the module is reloaded. In fact to augment the toplevel environment of the module does not change the exportations of the module and thus makes invisible to the outside these new bindings. Furthermore these new bindings cannot affect the body of the current module, they must be considered as grafted onto the module. Fortunately for code generation, the ex/importation clauses that set the mutability of the binding can be used for inlining or constant propagation even if the module uses `eval/cm`.

The modified equations for reference and assignment now test if the variable is defined before taking an action. The following equations are given in terms of $\mathcal{E}$ rather than $\mathcal{E}'$ since it is more readable.

$$\begin{array}{ll}
\mathcal{E}[\![\nu]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m = & \textbf{if } \rho_m(\nu) = \textit{lexenv-location-not-found} \\
& \textbf{then } \ \textbf{if } (\delta_m(\nu_m))(\nu) = \textit{lexenv-location-not-found} \\
& \qquad \textit{wrong}(\text{“}\texttt{Inexistent variable}\text{”}) \\
& \qquad \ \textbf{then } \kappa_m(\sigma_m(\delta_m(\nu_m)(\nu)), \sigma_m, \delta_m) \\
& \qquad \ \textbf{endif} \\
& \textbf{else } \kappa_m(\sigma_m(\rho_m(\nu)), \sigma_m, \delta_m) \\
& \textbf{endif}
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{E}[\![\texttt{(setq }\nu\ \pi\texttt{)}]\!]\nu_m\mu_m\rho_m\sigma_m\delta_m\kappa_m =} \\
\quad \mathcal{E}[\![\pi]\!](\nu_m, \mu_m, \rho_m, \sigma_m, \delta_m,\ \lambda\ \varepsilon_c\sigma_c\delta_c\ . & \textbf{if } \rho_m(\nu) = \textit{lexenv-location-not-found} \\
& \textbf{then } \kappa_m(\varepsilon_c, \sigma_c[\alpha \to \varepsilon_c], \delta_c[\nu_m \to \delta_c(\nu_m)[\nu \to \alpha]],) \\
& \qquad \textbf{where } \alpha = \textit{new-location}(\sigma_c, 1)\ ) \\
& \textbf{else } \kappa_m(\varepsilon_c, \sigma_c[\rho_m(\nu) \to \varepsilon_c], \delta_c)) \\
& \textbf{endif}
\end{array}$$

A variable is first looked up in the lexical environment ρ_m then in the toplevel module environment $\delta(\nu_m)$. Note that this semantics differentiates two erroneous situations: when a variable does not exist or when it exists but is undefined.

4.2 The Language of `eval/cm`

What kind of expressions can be submitted to `eval/cm` ? Since `eval/cm` is tied to a module, the syntax of its argument should be that of the module. This is particularly clear with respect to the variables defined at the toplevel of the module but is unclear with respect to the macros that were available at the point where `eval/cm` appears. Since we do not want macros to be kept at run-time we restrict the argument of `eval/cm` to be a pure expression of the language still able to use all the variables of the module. If one wants to use a special syntax before submitting expressions to **eval/cm**, then one has to call explicitely the necessary macro-walkers before invoking **eval/cm**.

An interesting application is for a module which defines a toplevel loop. Consider for instance: `(defmodule toplevel-`

```
    (expose library)
    (immutable (only (start) (expose toplevel-loop)))
  (set! start (lambda ()
                (display "? ")(newline)
                (display (eval/cm (read)))
                (newline)
                (start) )) )
```

The **start** entry point in the module **toplevel-loop** starts an interactive loop requiring an expression from the user and printing its value. The expression must be in the pure language and can use all the library as well as the **start** function. All evaluations will be performed in the visibility of this module and cannot alter **start** because of its immutability but allow to augment the module with new variables which will be mutable. Note that these new bindings can also alter the mutable locations imported from the **library** if any. Therefore all specialized modules like the **format** module (see Common Lisp) can have been compiled with the higher level of optimization since not containing call to **eval/cm**, they cannot be altered in a blind way.

To submit expressions in the pure language is rather crude. One can dream to benefit from some macros. The fix is simple although it will imply to load at run-time the modules containing the needed macros. We just write: `(defmodule toplevel-loop-allowing-some-macros`

```
    (union (expose library) (expose standard-macros))
    (immutable (only (start) (expose toplevel-loop-allowing-some-macros)))
  (let ((walk standard-macroexpand))
    (set! start (lambda ()
                  (display "? ")(newline)
                  (display (eval/cm (walk (read) walk)))
                  (newline)
                  (start) )) ) )
```

The standard walker is given by the value of **standard-macroexpand** from the **standard-macros** module and is the composition of all specialized walkers for the usual macros: **let**, **letrec**, **case** ... all the so-called *derived syntaxes* of R^3RS [Rees & Clinger 1986].

4.3 Local Macros

Another interesting application is local macros. In many Lisp systems it is possible to define a macro and to use it after it was defined. The following module allows that: **local-defmacro** is a normal macro that evaluates the associated expander in the module where **local-defmacro** is defined and extends the syntax to now know this new expander. This module is :`(defmodule local-defmacro`

```
    (union (expose standard) (expose standard-macros))
    (syntax (only (local-defmacro) (expose extended-syntax)))
  ;;(local-defmacro name (variables..) body)
  (set! local-defmacro
      (lambda (syntax)
        (letrec ((local-macros
                   (list (cons 'local-defmacro
                           (lambda (e m)
                             (if (and (pair? (cdr e))
                                      (pair? (cddr e)) )
                               (let ((name (cadr e))
                                     (variables (caddr e))
                                     (body (cdddr e)) )
                                 (set! local-macros
                                       (cons
                                        (cons name
                                              (let ((fun (eval/cm (standard-macroexpand
                                                                   `(lambda ,variables . ,body)
                                                                   standard-macroexpand ))))
                                                (lambda (e m)
                                                  (m (apply fun (cdr e)) m) ) ) )
                                       local-macros ) )
                                  `#t )
                                (error 'local-defmacro-syntax) ) ) )) ))
          (lambda (e m)
            (if (pair? e)
              (let ((expander (assq (car e) local-macros)))
                ((if expander (cdr expander) syntax) e m) )
              (syntax e m) ) ) ) ) )
```

The `local-defmacro` syntax modifier introduces a new syntax which on each form it walks, check if its `car` is the name of a local macro and expands this form if this is true; otherwise it looks like the previous syntax. Local macros are introduced by the first local macro which name is `local-defmacro`. Let us give an example of this module: `(defmodule use-of-local-defmacro`

```
    (union (expose library) (expose local-defmacro))
    (union ...)
  (local-defmacro list args
    (if (pair? args)
        `(cons ,(car args) (list . ,(cdr args)))
        `'() ) )
  (set! names (list 'car 'cdr 'cadr ...))
  ... )
```

An useful macro named `list` was defined and then used in the rest of the module. The language of the expander of `list` is of course the language allowed in the `local-defmacro-module` i.e.the pure language augmented with the standard macros.

4.4 Conclusions on Evaluation

We think that three main problems exist with respect to evaluation. The first one it tied to denotatonial semantics since to introduce `eval` violates compositionality, one of the tenets of denotational semantics. Since we saw that to introduce macros already enforces us to devise $\mathcal{E}'$, a semantical interpreter of values in store, the evil was already done. The second problem concerns the meaning of `eval`. We think that `eval/cm` best fits with our module proposal and is not far from the usual practices. The third problem is related to the cost of `eval/cm` since it involves keeping an evaluator in any applications using a module containing a reference to `eval/cm` plus the symbol tables of the toplevel environment of these modules. An evaluator can range from a clever compiler (4 MBytes) to a modest interpreter (50 kBytes). Symbol tables are generally never stripped off since the debugger needs them and the confusion between a variable and a symbol is always there in implementations. We thus think that the cost is not too much since probably the major use of `eval/cm` will be for toplevel loops and advanced macro programming.

On the negative side we will mention the very nature of `eval/cm` which acts like a function and is denoted as a function thanks to the parameter containing the name of the current module in every denotations. `eval/cm` may also be viewed as a keyword which meaning varies from module to module since it captures a different toplevel environment at each time. The distinction is subtle but real since a keyword is only syntax while a function is a first class object. In this latter case one may wonder what is the result of:

```
(defmodule wonder
    (...)
    (...)
    (eq eval/cm                ;is eval/cm
        (progn (loadmodule M)  ;always equal to
               eval/cm ) ) )   ;itself ?
```

5 Conclusions

We define a primitive module system with a clear semantics on top of a simple Lisp-like language. Fortunately, although primitive, it seems to embody all the right operations for a convenient practical model after dressing in new syntax. The model also extends comfortably to accomodate macros and a precise semantics for macros with predictable effects. We also show that the model can be extended to support the `eval` facility, which can then be given a precise meaning.

6 Acknowledgments

Much of this work has been done in the spirit of EuLisp and we thank all of the members of that group. We also wish to acknowledge Bernard Serpette and our students for numerous and fruitful discussions in which modules and macros have revealed their hidden faces. We also wish to thank all the referees who drove us in a better way.

References

[Bawden & Rees, 1988] Alan Bawden, Jonathan Rees, *Syntactic Closures*, Proceedings of 1988 ACM Conference on Lisp and Functional Programming, pp 86-95, ACM Press, New York, 1988.

[CAML, 1989] *CAML Reference Manual*, INRIA, 1989.

[Curtis, 1990] Curtis P. & James Rauen, Lisp conf 90.

[Dybvig 87] *The Scheme Programming Language*, Prentice-Hall, 1987.

[Dybvig *et al.*, 1988] R. Kent Dybvig, Daniel P. Friedman, Christopher T. Haynes, *Expansion-Passing-Style: A General Macro Mechanism*, Lisp and Symbolic Computation, Vol. 1, No. 1, June 1988, pp 53-76.

[Hudak, Wadler *et al.*, 1988] Hudak P. & Wadler P., (eds.) *Report on the Functional Programming Language Haskell*, Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-666, December 1988.

[Kelsey & Hudak, 1989] Kelsey R. & Hudak P., *Realistic Compilation by Program Transformation* Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, ACM Press, New York, 1989.

[Kohlbecker *et al.*, 1986] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, Bruce Duba, *Hygienic Macro Expansion*, Proceedings of 1986 ACM Conference on Lisp and Functional Programming, pp 151–161, ACM Press, New York, 1986.

[MacQueen, 1988] MacQueen D., *Modules for Standard ML*, Proceedings of 1988 ACM Conference on Lisp and Functional Programming, ACM Press, New York, 1988.

[Matthews, 1983] Matthews D.C.J, *Programming Language Design with Polymorphism*, University of Cambridge Computer, Laboratory Technical Report No. 49, 1983.

[Moon, 1988] Moon D., Private communication.

[Muchnick & Pleban, 1980] Steven S. Muchnick, Uwe F. Pleban, *A Semantic comparison of Lisp and Scheme*, Conference Record of the 1980 Lisp Conference, pp 56–64.

[Pitman, 1980] Kent M. Pitman, *Special Forms in Lisp*, Conference Record of the 1980 Lisp Conference, pp 179–187.

[Queinnec 90] Christian Queinnec, *Struggle, the First Denotational Game*, Proceedings of EuroPaL'90, Cambridge, UK, March 1990, pp 351-361.

[Rees *et al.*, 1986] Rees J.A., *et al*, *The T Manual*, YALEU Technical Report, 1986.

[Rees & Clinger 1986] Jonathan A. Rees, William Clinger, *Revised*^{3} *Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37–79, ACM Press, New York, 1986

[Slade *et al*, 1987] Slade S., *et al*, *The T Programming Language, a Dialect of Lisp*, Prentice-Hall 1987.

[Steele 90] Steele G.L. Jr., *Common Lisp the language*, 2nd edition, Digital Press, 1990.

[Stoy, 1977] Stoy J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[Symbolics] Symbolics Genera 7 Documentation.

[Texas Instruments] PC-Scheme Documentation, Texas Instruments.

A Semantic Utility Functions

$\rho_{init} = \lambda\ \nu\ .\ \textit{lexenv-location-not-found}$ — *The initial lexical environment*

$\delta_{init} = \lambda\ \nu\ .\ \textit{not-loaded}$ — *The initial dynamic load environment*

$\mu_{init} = \lambda\ \nu\ .\ \textit{modenv-module-not-found}$ — *The initial module environment*

```
$\mathcal{CE} = \lambda\ \nu^* \nu_f^* \nu_l^*\ .$ $\textbf{if}\ \nu^* = <>$ — Check exportations
        $\textbf{then}\ \textit{true}$
        $\textbf{else}\ (\nu^* \downarrow 0 \in \nu_f^*) \vee (\nu^* \downarrow 0 \in \nu_l^*) \wedge \mathcal{CE}[\![\nu^* \dagger 1]\!]$
        $\textbf{endif}$
```

$\mathcal{CI}[\![(\ \nu_m\ \nu_r\ \nu_l)\ \iota^*]\!] =$ — *Check importations*
$\lambda\ \nu_m \nu_r \nu_l \iota^*\ .\ \lambda\ \mu\ .\ \neg(\mu(\nu_m) = \textit{modenv-module-not-found}) \wedge (\nu_r \in \mu(\nu_m) \downarrow 0) \wedge (\mathcal{CI}[\![\iota^*]\!])(\mu)$

$\mathcal{CI}[\![]\!] = \lambda\ \mu\ .\ \textit{true}$

$\mathcal{L}[\![\ ]\!] : \mathbf{ModEnv} \times \mathbf{Store} \times \mathbf{LoadEnv} \times \mathbf{Cont} \rightarrow \mathbf{Answer}$ — *load imported modules*

```
$\mathcal{L}[\![(\ \nu_m\ \nu_r\ \nu_l)\ \iota^*]\!] =$
$\lambda\ \nu_m \nu_r \nu_l \iota^*\ .\ \lambda\ \mu\sigma\delta\kappa\ .$ $\textbf{let}\ \theta = \lambda\ \varepsilon_c \sigma_c \delta_c\ .\ (\mathcal{L}[\![\iota^*]\!])(\mu, \sigma_c, \delta_c, \kappa)$
        $\textbf{in}\ \ \textbf{if}\ \delta(\nu_m) = \textit{not-loaded}$
            $\textbf{then}\ (\mu(\nu_m) \downarrow 1)(\sigma, \delta, \theta)$
            $\textbf{else}\ \theta((\textit{boolean-true})_{\mathbf{Bool}} \uparrow^{\mathbf{Value}}, \sigma, \delta)$
            $\textbf{endif}$
```

$\mathcal{L}[\![]\!] = \lambda\ \mu\sigma\delta\kappa\ .\ \kappa((\textit{boolean-true})_{\mathbf{Bool}} \uparrow^{\mathbf{Value}}, \sigma, \delta)$

$\mathcal{LV}[\![\ ]\!] : \mathbf{Id}^\star$ — *extract local names out of importations*

$\mathcal{LV}[\![(\ \nu_m\ \nu_r\ \nu_l)\ \iota^*]\!] = \lambda\ \nu_m \nu_r \nu_l \iota^*\ .\ < \nu_l > \S \mathcal{LV}[\![\iota^*]\!]$

$\mathcal{LV}[\![]\!] = <>$

$\mathcal{R}[\![(\ \nu_m\ \nu_r\ \nu_l)\ \iota^*]\!] = \lambda\ \nu_m \nu_r \nu_l \iota^*\ .\ \lambda\ \delta\ .\ (\mathcal{R}[\![\iota^*]\!])(\delta)[\nu_l \rightarrow (\delta(\nu_m))(\nu_r)]$ — *Retrieve imported bindings*

$\mathcal{R}[\![]\!] = \lambda\ \delta\ .\ \rho_{init}$

```
$augment - locations =$ — Augment the toplevel environment with the proper part
$\lambda\ \nu_f^* \rho\sigma\theta\ .$ $\textbf{if}\ \nu_f^* = <>$
    $\textbf{then}\ \theta(\rho, \sigma)$
    $\textbf{else}\ \ \textbf{if}\ \rho(\nu_f^* \downarrow 0) = \textit{lexenv-location-not-found}$
        $\textbf{then}\ \ \textbf{let}\ \alpha = new - location(\sigma)$
            $\textbf{in}\ augment - locations(\nu_f^* \dagger 1, \rho[\nu_f^* \downarrow 0 \rightarrow \alpha], \sigma[\alpha \rightarrow \textit{store-undefined-value}], \theta)$
        $\textbf{else}\ augment - locations(\nu_f^* \dagger 1, \rho, \sigma, \theta)$
        $\textbf{endif}$
    $\textbf{endif}$
```

$\mathcal{FV}$: not given — *Extracts free variables out of a form*

new-location, *new-locations*: implementation dependent