Modules, Macros and Lisp

Eleventh International Conference of the Chilean Computer Science Society, oct 1991, Santiago (Chile), Plenum Publishing Corporation, New York NY (USA), pp 111-123.

Christian Queinnec[1*] and Julian Padget[2†]

[1] École Polytechnique & INRIA–Rocquencourt, France, queinnec@polytechnique.fr
[2] University of Bath, United Kingdom, padget@maths.bath.ac.uk

Abstract

Many modern languages offer a concept of modules to encapsulate a set of definitions and make some of them visible from outside. Associated to modules are directives ruling importations and exportations as well as controlling other capabilities like renaming or qualified notation. We propose here a language to handle sets of named locations with the aim of describing precisely which locations are to be shared between modules and under which local names they are known. The language is symmetric for imports and exports. It is also extensible since it provides a framework to handle named locations. Mutability, for instance, can readily be expressed thanks to a simple extension of our language.

Another more subtle extension is to offer a protocol for modules with "macros". Macros allow a user to extend the syntax of a language by providing rewriting rules expressed as computations performed on the representation of programs. Lisp is a language offering macros and much of its extensibility stems from them. However, many problems arise from macros, amongst them being the exact definition of when, where and how the macroexpansion is done to turn the body of a module using extended syntaxes into a regular form in the bare, unextended language.

This paper presents, in the framework of Lisp, a definition of modules offering tight control over module environments as well as a precise semantics of separate compilation and macroexpansion. These new (for Lisp) capabilities definitely add some power relative to the management of name spaces but also justify compiling optimizations (inlining, partial evaluation etc.) and, above all, turn Lisp into a language suitable for application delivery since applications are finite collections of modules. Our proposal contributes to a vision of Lisp composed of a multitude of linked and reusable modules. Our results concerning name space management and macros can also be applied to other languages.

*This work has been partially funded by Greco de Programmation.
†This work has been partially funded by the Science and Engineering Research Council.

Two different implementations of these concepts exist: in FEEL, the implementation of EuLisp done by the University of Bath and in the idiom of Icsla, a dialect of Lisp designed at INRIA-Rocquencourt and École Polytechnique. This paper reflects the results of the experiments carried out in these systems during the last year.

1 Introduction

A module encapsulates a set of definitions. Some of these definitions must be made available to clients of the module while others must remain invisible. These definitions can be shared by a number of different methods: name, location or value. Names is the method used in Common Lisp[Steele, 1990] and implies the existence of a unique name space where two entities from two different modules and bearing a same name cannot be present together. Values is the method used, for example, in Poly [Matthews, 1983] but is of limited worth since this makes modules little different from closures. As in more classical languages, such as C, we feel locations are the right thing to exchange in presence of assignments since they hold (and allow access to) values, but are independent of names and therefore can be alpha-converted on a local basis.

The exportation language must be able to handle sets of locations and to perform the usual set operations on them. Since a location is known by its name within a module, the exportation language must then handle locations through their names. However, there is a restriction which is that no two different locations may have the same name, a situation introducing ambiguity known as *name clash*. Since the programmer's naming convention usually has some semantic import for the value being manipulated, the default behaviour is to maintain the association between the name given and the location when it was defined. Conversely since a location is independent of its name, this association may be changed.

Similarly, the importation language, whether importing locations under their real names or new names, must have equivalent power. In consequence, we have developed a common language for both importation and exportation. The current syntax of this language, as presented below, was chosen to be regular rather than convenient. Abbreviations can be contrived, but are irrelevant to this paper.

A module of Lisp can also contain executable code which will be evaluated when the module is loaded. Hence it is necessary to define very carefully how a module is processed and loaded. The information stored in the importation and exportation directives shows a given module depends on other modules. This dependency shows itself both when processing and loading a module: the modules on which it depends must be processed or loaded beforehand. Introducing macros imposes the additional constraint that to obtain the pure code for the body of a module, a computation is required. Finally, we define how to start a computation, that is, how to invoke a function within a module.

Section 2 introduces modules and how they are processed. Section 3 focusses on the importation and exportation language. An example of an extension, to handle mutability of locations, appears in section 4. The second extension, macros, is presented in section 5. Various examples are given throughout the paper. The results of practical experience with this model are covered in the conclusions.

2 Module

A module comprises four parts: a name, an importation directive, an exportation directive and a body[1]. Whilst it may not be the most suitable form in practice, we will assume the representation of a module in this paper to be a file with the same name as the module, the first two expressions contained therein being the import and export directives respectively,

[1]For convenience, we suppose a language for the body which is regular Scheme augmented with an `access` special form which will be explained later.

followed by the body. The language we use is lexically scoped, but it is convenient for us to be able to distinguish between the bindings made within a program—such as by `lambda`—and the bindings created by `define` or by importation. We call the former *inner-lexical* bindings and the latter *top-lexical* bindings. Where there is no ambiguity, we simply refer to *lexical* bindings. As an example, the following module defines the fibonacci function:

```
;;; ====== The fibonacci module ======
(expose standard)

(union
  (immutable (rename ((fibonacci bm-fib))
               (except (fib20 fib-max)
                 (only-from (fibonacci) (expose fibonacci))) ))
  (mutable (qualified (only (fib-max) (expose fibonacci)))) )
;;; ======         ======

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

(define fib20 (fib 20)) ;better precompute it !

(define (bm-fib n)
  (cond
    ((= n 20) fib20)
    ((> n fib-max) (error "Too big number" n))
    (else (fib n))))
```

A module is processed as follows:

1. The importation directive is processed in order to build the imported part of the top-lexical environment in which to analyse the body of the module.

   For instance, in the above module, the functions `<`, `+`, `,` `=`, `error` and `>` are, among others, imported from the module `standard`.

2. The body of the module is analysed: all free variables are extracted from it and these define the proper part of the top-lexical environment. The combination of the imported part and the proper part forms the top-lexical environment of the module currently being processed. We will refine this phase when considering macros in section 5.

   In the above module, the variables `fib`, `fib20`, `bm-fib` and `fib-max` are free. They are thus considered to form the proper part of the top-lexical environment of module `fibonacci`.

3. The exportation directive is processed to determine the exportations.

   In the above module, only the locations `fib`, `bm-fib` and `fib-max` are exported. The first two are exported as immutable locations: they cannot be assigned to from outside, whereas `fib-max` can. The location `bm-fib` is exported under a new and more meaningful name: `fibonacci`.

4. Finally, the module is then added to the module environment (a disjoint environment). The "real compilation" is done here since anything that can affect the meaning or the behaviour of the module is now known.

   The module environment associates names to processed modules. A processed module comprises the information about its dependencies, importations and exportations. The processed module is presumably also associated to its compiled body, top-lexical environment for debugging, amongst other things. This module environment does not need to be represented within the programming language environment, but can be based on the file system of the underlying operating system.

After a module has been processed it is ready to be imported or loaded. A processed module is said to be dependent on other modules if it imports or exports locations from them. A module, m, is loaded as follows:

1. the modules on which m depends are loaded, but modules already loaded are not reloaded.

   These modules either contribute to the imported part of the top-lexical environment of the module or may accompany this module and provide any client of m with the accompanying facilities.
2. the proper part of the top-lexical environment of m is allocated.

   Now, the top-lexical environment of the module is complete.
3. the body of the module is evaluated sequentially in this top-lexical environment.

   The top-lexical environment can be viewed as a kind of "`letrec`" form allowing mutual recursive definitions. Note that in the example above, the variable `fib20` will be computed at load-time so that faster answers will result when calling the `bm-fib` function.

After a module is loaded, any location exported from it can be used to start a computation provided it holds a function that can be applied to the arguments of the intended computation. The means to start a computation can be supplied from Lisp itself, or from the shell of the underlying operating system. In this instance, we suppose ourselves to be operating on top of UNIX[2]where we have defined the `start-module` command to start a computation. Given the above module, we can submit numbers[3] to the `fib` function:

```
% start-module fibonacci fib 5 ; start-module fibonacci fib 10
589
```

Starting the `fibonacci` module forces the `standard` module (and all the modules on which it depends) to be loaded before the contents of the `fib` location, a monadic function, can be invoked. An error would be signalled if the number of submitted arguments does agree with the arity of the specified entry point.

3 Importations and exportations directives

We give a grammar for import export directives in figure 1.

directive	::=	`(expose` *module-name*`)`
	|	`(only (`*name* `...)` *directive*`)`
	|	`(except (`*name* `...)` *directive*`)`
	|	`(only-from (`*module-name* `...)` *directive*`)`
	|	`(rename (`*substitution* `...)` *directive*`)`
	|	`(qualified` *directive*`)`
	|	`(union` *directive*`...)`
substitution	::=	`(`*new-alias old-alias*`)`

Figure 1: Grammar of basic directives

The language is completely symmetric with respect to importation or exportation except in the case of the `expose` directive. The `expose` directive takes the name of a module and

[2] UNIX is a trademark of AT&T.

[3] We suppose that arguments given from the shell are submitted as if read by Lisp, `fib` is then invoked on the number ten and not the string `"10"`. We also suppose that the final result is printed according to Lisp rules. These assumptions are false in real implementations but simplify the examples.

produces the set of tuples describing all its exported locations. For the moment, these are quadruples as follows:

$$< \mathit{alias}, \mathit{name}, \mathit{module\text{-}name}, \alpha >$$

Such a tuple represents the location α in the proper part of the top-lexical environment of the module named *module-name*, this location being known as *name* and referred to as *alias*. Initially *name* and *alias* are the same. The `standard` module produces a large set of tuples describing all the usual functions that are considered as standard in Scheme such as `cons`, `null?` and so on, as well as the arithmetic functions `+`, `-` amongst others.

Of course, it does not make sense to `expose` the module currently being processed in an importation directive. However, this is reasonable at exportation since its top-lexical environment is completely defined. Thus, `expose`-ing the current module for exportation produces the set of all locations belonging to its top-lexical environment.

Sets of tuples representing locations can be filtered to deliver some subset and the filtering can be done on the name or on the module-name stored in the tuple. The `only` directive accepts the set produced by the enclosed *directive* and only those locations which have an alias belonging to the list of *name* ... are allowed to pass through. The `except` directive allows only those locations whose alias does *not* belong to the list of *name* ... to pass through. The `only-from` directive permits only those locations which are defined in the list of modules *module-name* ... to pass. For the sake of symmetry one can easily imagine the behaviour of the `except-from` directive.

Sets of tuples can be merged by means of the `union` directive provided no name clash occurs. Two tuples are different if they mention different *module-names* or different *locations*. A name clash occurs when two different tuples have the same alias. Note that it is harmless to merge a set of tuples with itself: `(union (expose M) (expose M))` is the same as `(expose M)`.

One way to resolve a name clash is to rename one of the offending tuples. Renaming can be achieved using the `rename` directive. The `rename` directive takes as input a set of tuples and produces a new set of tuples where any original tuple having an alias equal to one of the *old-alias* is rewritten as a new tuple with the associated *new-alias*. All other tuples are unchanged. Note that all directives are functional and do not modify tuples.

It is also interesting in many cases to offer the visiblity of a location without providing any name for it. Modula offers a qualified notation which allows a kind of "absolute name" for a location. We add to the language of module bodies the `access` special form for that purpose: `(access` *module-name alias*`)` refers to the location which was exported under *alias* from *module-name*[4]. A location can be referred to using qualified notation only if it is exported, hence `access` forms cannot break the encapsulation provided by a module. When a location is exported (or imported) as qualified, no alias is associated to it so it can only be referred to using qualified notation.

Directive example

Suppose that the exportation directive of the `fibonacci` module was in fact:

```
(union (rename ((fibonacci bm-fib))
         (except (fib20 fib-max)
           (only-from (fibonacci) (expose fibonacci))) )
       (qualified (only (fib-max) (expose fibonacci))) )
```

The directive `(expose fibonacci)` produces the set of all locations of the top-lexical environment of the `fibonacci` module. This set contains:

[4]Note that contrary to the definition of tuples, it is not required that *alias* names a location belonging to the proper part of *module-name*. It may also belong to the imported part of the top-lexical environment and then reexported to be visible. Also note that the access special form can be implicitely written with a qualified name such as *module-name*: *name*, if so desired.

$\{ \quad < \texttt{fib}, \texttt{fib}, \texttt{fibonacci}, \alpha_0 >$
$\quad < \texttt{fib20}, \texttt{fib20}, \texttt{fibonacci}, \alpha_1 >$
$\quad < \texttt{bm-fib}, \texttt{bm-fib}, \texttt{fibonacci}, \alpha_2 >$
$\quad < \texttt{fib-max}, \texttt{fib-max}, \texttt{fibonacci}, \alpha_3 >$
$\quad < \texttt{+}, \texttt{+}, \texttt{standard}, \alpha_{37} >$
$\quad < \texttt{cons}, \texttt{cons}, \texttt{standard}, \alpha_{21} >$
$\quad \ldots \qquad \}$

Note that since all locations from `standard` were imported, they all belong to the top-lexical environment of the `fibonacci` module. The `only-from` directive filters out of the above set the proper locations of the `fibonacci` module i.e.

$\{ \quad < \texttt{fib}, \texttt{fib}, \texttt{fibonacci}, \alpha_0 >$
$\quad < \texttt{fib20}, \texttt{fib20}, \texttt{fibonacci}, \alpha_1 >$
$\quad < \texttt{bm-fib}, \texttt{bm-fib}, \texttt{fibonacci}, \alpha_2 >$
$\quad < \texttt{fib-max}, \texttt{fib-max}, \texttt{fibonacci}, \alpha_3 > \quad \}$

From which are removed locations `fib20` and `fib-max`. After that, `bm-fib` is renamed as `fibonacci` thus yielding:

$\{ \quad < \texttt{fib}, \texttt{fib}, \texttt{fibonacci}, \alpha_0 >$
$\quad < \texttt{fibonacci}, \texttt{bm-fib}, \texttt{fibonacci}, \alpha_2 > \quad \}$

And finally, the qualified `fib-max` location is added, but with a null alias. The final result is thus:

$\{ \quad < \texttt{fib}, \texttt{fib}, \texttt{fibonacci}, \alpha_0 >$
$\quad < \texttt{fibonacci}, \texttt{bm-fib}, \texttt{fibonacci}, \alpha_2 >$
$\quad < , \texttt{fib-max}, \texttt{fibonacci}, \alpha_3 > \qquad \}$

Another directive example

The `standard` module itself is, presumably, not a monolithic module but a composition of more primitive modules gathered in a single module so that the user has only one module to import rather than many. Our directive language allows us to write modules whose only effect is to collect locations in a single module: linkers or partial evaluators can benefit from this situation. A first way to define `standard` would be to import all needed locations then to reexport them in one go:

```
;;; ====== The standard module (first attempt) ======
(union (expose arithmetic) (expose pair) ...)

(expose standard)
;;; ======                ======
;;; No body at all
```

A second, less naïve, approach is to gather all needed locations at exportation time:

```
;;; ====== The standard module (second attempt) ======
(union)

(union (expose arithmetic) (expose pair) ...)
;;; ======                ======
;;; No body at all
```

It might seem strange to be exporting locations which were not imported, but this facility contributes to the provision of more useful dependency information as will be shown with macros in section 5. In the previous example, the `standard` module depends (at least) on the modules `arithmetic` and `pair`.

4 Mutability

When exporting locations, the writer of a module may provide immutable definitions and also some special locations which can be changed by the user. In our preferred example, the `fib-max` location must be set by the client of the `fibonacci` module before s/he invokes the `fibonacci` function otherwise an "uninitialized location" error would be signalled. Part of the interface is the explicit mutability of this location. Making other locations immutable for clients denies them the right to modify the contents of these locations. The processor can take advantage of that and can analyse the body of the module to determine which locations are initialized for certain and which locations are only assigned once and never again changed thereafter. The results of such analyses are fruitful since constants can be safely propagated and the invocation of the contents of immutable locations can be inlined, for instance.

We therefore want to specify the mutability of exported locations. First we extend the previous tuple definition to a quintuple with an additional field specifying the mutability of the location. Initially, tuples are created with no restrictions on mutability. Tuples become:

$$< \mathit{alias}, \mathit{name}, \mathit{module\text{-}name}, \alpha, \{\mathit{mutable} \mid \mathit{immutable}\} >$$

In consequence we add two new directives.

$$\begin{array}{lcl} \mathit{directive} & ::= & (\texttt{immutable}\ \mathit{directive}) \\ & | & (\texttt{mutable}\ \mathit{directive}) \end{array}$$

The `immutable` directive produces a new set of tuples with a mutability field set to *immutable*. The `mutable` directive is not really legitimate since an immutable location cannot revert safely to a mutable one. The `mutable` directive only serves to emphasize that a location is exported with write access.

The initial exportation directive of the `fibonacci` module therefore now yields the following set of tuples:

$$\{ \begin{array}{l} < \texttt{fib}, \texttt{fib}, \texttt{fibonacci}, \alpha_0, \mathit{immutable} > \\ < \texttt{fibonacci}, \texttt{bm-fib}, \texttt{fibonacci}, \alpha_2, \mathit{immutable} > \\ < , \texttt{fib-max}, \texttt{fibonacci}, \alpha_3, \mathit{mutable} > \end{array} \}$$

5 Macros

Macros permit the extension of the syntax of the language. A macroexpansion phase translates the body of a module into a regular expression without additional syntaxes. In Lisp, this translation might use all the resources of the language. Macros and macroexpansion present many problems which can be summarised in three words: *when*, *where* and *how*. We mainly focus here on the when and the where rather than how. Given the number of variants that exist, we prefer to offer a macroexpansion protocol allowing the user to decide how macros are represented and how module bodies are expanded. It is then possible to implement the proposal of [Clinger & Rees, 1991] or Expansion Passing Style (EPS) advocated in [Dybvig *et al.*, 1988] as legal implementations of the macroprotocol stated below.

Associated with each macro is a syntax expander, that is a function taking forms using the extended syntax and translating them into more verbose, simpler forms lacking that extension. While most macros are simple rewriting rules, some are very complex and embed specialised compilers: see for example the `loop` macro [Steele, 1990] or pattern matching [Queinnec, 1990]. Macros may also need an internal state like the `define-class` macro which has to manage the inheritance tree of classes. Since using a macro involves a computation, we impose the requirement that macroexpansion be implemented as a module, to be called the macroexpansion module, exporting a location named `macroexpand-module-body`. When it is necessary to macroexpand the body of a module, the macroexpansion module is loaded and the function held in the `macroexpand-module-body` location is invoked with the necessary arguments which are: (i) the body of the module and (ii) the complete syntax that might be used by the module. The result will become the macroexpanded body of the currently processed module.

The remaining problem is how to represent syntaxes. Syntaxes can be additive, allowing rewriting rules to be freely combined. Conversely some syntaxes might want to be locally exclusive. We therefore conclude that the macroexpansion module must export another location named `initial-syntax`, holding a function with a signature similar to `macroexpand-module-body` but whose rôle is to walk its first argument (the module body), check if there are some uses of extended syntaxes and, if so, expand them using its second argument (the syntax). If no macros occur then the macroexpanded body is equal to the initial body i.e.

`(macroexpand-module-body` *body* `initial-syntax)` $\rightarrow$ *body*

A macro is represented by a function which takes three arguments: (i) the current syntax, (ii) the name under which the macro is imported, (iii) the name of the module currently being processed. This function generally enriches the current syntax with a (or some) new rewriting rule(s) and returns a syntax that will be considered as the new current syntax. Hence macros do not exist *per se* but are represented by ordinary functions which "install" new syntaxes. We add a new directive to signal that some locations hold functions that must be considered as macros:

$$\textit{directive} \quad ::= \quad \texttt{(syntax } \textit{directive}\texttt{)}$$

To explain the `syntax` directive, we again extend tuples, now to become sextuples, with a field specifying whether the location is syntax or not:

$$< \textit{alias}, \textit{name}, \textit{module-name}, \alpha, \{\textit{mutable} \mid \textit{immutable}\}, \{\textit{regular} \mid \textit{syntax}\} >$$

The `syntax` directive takes a set of tuples and produces a new set of tuples with the syntax attribute set.

After processing an importation directive, imported locations are sorted into two sets: ordinary and syntax locations. These latter represent the set of syntaxes that might be used in the body of the module currently being processed. These partial syntaxes may then be composed into a single syntax. To macroexpand this body the following actions are taken:

1. the macroexpansion module is loaded and the current syntax is set to the contents of the `initial-syntax` location.

   Various macroexpansion modules can exist, but for now, we have implemented Expansion Passing Style (EPS) and COMMON LISP-like macros with hashtables.
2. For any syntactic location $<$ *alias*, *name*, *module-name*, α_i, *mutability*, *syntax* $>$, the module *module-name* is loaded, the contents of the location α_i is retrieved and applied to the current syntax, the *alias* under which the syntactic location was imported and the name of the module being processed. We remark that the set of macros is not ordered.

   This scheme allows macros to be renamed thus avoiding syntax clashes. The name of the currently processed module is also given so that macros define in the module being processed can generate `access` forms relative to it.
3. The original body of the processed module and the final syntax obtained in the previous phase are submitted to the contents of the `macroexpand-module-body` of the macroexpansion module. The result is the macroexpanded body.

   As with all computations, this operation might not terminate or might yield a syntactically recursive program, see [Queinnec & Padget, 1990] for further details.
4. all the syntactic locations are removed from the top-lexical environment of the module being processed.

   Nothing remains from macroexpansion if it is not needed at run-time.

To resume our discussion of the macroexpansion protocol:

- A module is processed with respect to a specific macroexpansion module. Such a module can be explicitly passed to the processor as a compiler option, assuming a reasonable default value.
- The macroexpansion module exports a location named `macroexpand-module-body` whose signature is:

  (`macroexpand-module-body` *module-body syntax*) → *macroexpanded-module-body*
- The macroexpansion module exports a location named `initial-syntax` whose signature is:

  (`macroexpand-module-body` *module-body syntax*) → *module-body*
- A macro is a location with the *syntax* attribute whose signature is:

  (*macro syntax keyword module-name*) → *syntax*

We remark that no constraints exist on the nature of syntaxes. They can be represented by functions as in EPS or using a list of hashtables to provide COMMON LISP-like facilities. A macro may introduce more than one syntax at once or even no syntax at all, but just side-effect the compilation environment. An essential macroexpansion module using EPS is given in the appendix.

Example

Suppose we wish to offer a `fibonacci` macro which when given a natural number expands into the correct fibonacci number. We can write the following `fibmac` module, using the `define-macro` macro of `standard` to hide implementation details[5]:

```
;;;====== The fibmac module ======
(union (expose standard) (expose fibonacci))

(union (syntax (only (fibonacci) (expose fibmac)))
       (qualified (only (fib) (expose fibonacci))) )
;;;======        ======
(define-macro (fibonacci exp)
  (if (integer? exp)
      (fib exp)
      `((access fibonacci fib) ,exp) ) )
```

This module defines a function which respects the macroexpansion protocol courtesy of the `define-macro` macro. The `fibonacci` location is exported as syntax explicitly by the user. Note that since the macro may be invoked on a number, the macro itself requires access to the `fib` function of the `fibonacci` module so this latter is exposed at importation in `fibmac`. Also note that the expansion of a call to the `fibonacci` macro might use the `fib` function of the `fibonacci` module and that therefore must be visible from any client of the `fibmac` module. Since it would be cumbersome to force the user of the `fibonacci` macro to remember to import the `fibonacci` module explicitly, we simply mention it in the exportation directive of `fibmac`. Thus, we completely hide the details of the `fibonacci` macro.

Let us give an example of a client of `fibmac`, say the `fibuser` module, which renames the `fibonacci` macro into `fib`:

```
;;;====== The fibuser module ======
(union (only (define list) (expose standard))
       (rename ((fib fibonacci)) (expose fibmac)) )

(only (test) (expose fibuser))
```

[5] See appendix.

```
;;;======        ======
(define (test arg)
  (list (fib 10) (fib arg)) )
```

A lot of standard syntaxes are exported from of `standard`, for brevity here we only show `define`. Two macros are therefore visible within `fibuser`. The set of locations yielded by the importation directive is:

{ < `define`, `define`, `standard`, α_{80}, *immutable*, *syntax* >
< `list`, `list`, `standard`, α_{117}, *immutable*, *regular* >
< `fib`, `fibonacci`, `fibmac`, α_{831}, *immutable*, *syntax* >
< , `fib`, `fibonacci`, α_0, *immutable*, *regular* > }

The official macroexpand module (say `macroexpand`) and modules `standard`, `fibmac` and `fibonacci` (on which `fibmac` depends) are then loaded since some macros in these might be used. The complete syntax that will be used to macrowalk the body of `fibuser` is thus the result of either:

```
((access fibmac fibonacci)
 ((access standard define)
  (access macroexpand initial-syntax)
  'define 'fibuser )
 'fib 'fibuser )
```

```
((access standard define)
 ((access fibmac fibonacci)
  (access macroexpand initial-syntax)
  'fib 'fibuser )
 'define 'fibuser )
```

Since macros defined by `define-macro` are additive, the two previous forms give equivalent syntaxes, that is, the order is immaterial. This final syntax is then used for macroexpanding the body of `fibuser`. The macroexpanded body is therefore the result of:

```
((access macroexpand macroexpand-module-body)
 `(begin (define (test arg)                         ;the initial body
           (list (fibonacci 10) (fibonacci arg)) ))
  ((access fibmac fibonacci)                        ;the proper name
   ((access standard define)
    (access macroexpand initial-syntax)
    'define 'fibuser )
   'fib 'fibuser ) )                                ;the alias
```

The result, that is, the macroexpanded body, is:

```
(set! test (lambda (arg)
             (list 89 ((access fibonacci fib) arg)) ))
```

The generated `access` form is legal since it refers to a visible location which has been imported thanks to the importation of the `fib` macro. After macroexpansion, the top-lexical environment is shrunk to the set of ordinary locations:

{ < `list`, `list`, `standard`, α_{117}, *immutable*, *regular* >
< , `fib`, `fibonacci`, α_{831}, *immutable*, *regular* > }

It is then clear that the `fibuser` module only depends on modules `standard` and `fibonacci` at run-time and no longer on `fibmac`. This last module has been used for syntax only and has disappeared from the run-time.

6 Related Work

Traditional solutions for information hiding in Lisp were supported by read-time symbol space management (the packages of Common Lisp). These facilities are difficult to master, can be bypassed and were not designed to assist with separate compilation. The main work on modules was presented in [Curtis & Rauen, 1990]. Many differences exist between that proposal and ours. In particular, module bodies and interfaces are separated in Curtis's proposal. Although following software engineering principles, we think that the interface can be deduced from the module body and its exportation directive, that some optimizations such as inlining cannot be done if the module body is not provided and finally that their separation does not fit harmoniously with macros. Macros cannot be used, nor exported, if unknown. Complex macros cannot be entirely put in the interfaces with all the definition of the utilities they use which needs to remain hidden. We therefore decided not to invent new concepts but just to define macros founded on simple modules.

Another big difference is that macros involve computations and therefore, the order in which macroexpansions are performed matters. The vast majority of macros are very simple rewriting rules that can be expressed in the restricted pattern language of [Clinger & Rees, 1991] and which do not depend on the exact order of macroexpansions. However complex macros (such as `define-class`) are not amenable to this view yet we still to be able to define them within the module framework. We therefore choose (i) a sequential evaluation semantics for module bodies and (ii) to let the user defines his/her own macroexpansion model. It is then possible to adopt the best of all the solutions to the different problems macros create.

The language we describe differs from other implementations of Lisp or Scheme since it is only a language of modules and not a language of expressions submitted to an interactive top-loop. We describe in [Queinnec & Padget, 1990] an evaluation feature allowing for the definition of modules defining new top levels. This point of view is rather novel for Lisp yet does not restrict its power. Autonomous applications as collections of linked and reusable modules can now be conceived.

7 Conclusions

We presented in this paper simple but useful modules that allow the precise definition of how to share locations. We designed a language to handle sets of named locations which can be used both for importations and exportations. We extended it straightforwardly to deal with mutability.

We defined how modules are processed, loaded and started as well as how collections of modules can be considered as complete applications that may be run autonomously from the underlying operating system.

We proposed a macroexpansion protocol which allows the parameterisation of a compiler giving it the power of macros while retaining a simple semantics. Our protocol does not enforce a particular macroexpansion algorithm nor the exact representation of macros. We therefore complement proposals such as [Dybvig *et al.*, 1988] or [Clinger & Rees, 1991]. We also give additional power to the user to rename macros.

Two implementations of modules using these ideas have been carried out, one at the University of Bath and another at INRIA-Rocquencourt and École Polytechnique. The construction of the `standard` module (which is a single module offering all the needed syntaxes and usual functions) makes use of all the features presented above: (i) renaming can be used, for instance, to export simultaneously `call/cc` and `call-with-current-continuation` as two aliased names for the same location. An assignment to one of these locations is guaranteed to be seen from the other. (ii) Modules form a tree. Mutually recursive functions that are sufficiently important to be defined in separate modules are defined as follows: first a module defines the two locations of the two mutually recursive functions and mutably exports them, the two functions are then defined in their own modules by assignment and eventually these two modules are gathered in a single module. The two initial locations can now be exported as immutable from this last module. (iii) Our module scheme eases interoperability and linking modules to existing C libraries. More complex macros such as `define-class` have also been written and this experience lead to a careful examination of what belongs to macroexpansion-time, load-time and to run-time. To merge syntactic locations with regular locations makes importations of complex macros easier for the user since all irrelevant details dealing with module dependencies are hidden.

We are now considering more interesting analyses such as partially evaluating modules which just collect other modules. A formal denotational semantics of our proposal was out of the scope of this paper but is fully detailed in [Queinnec & Padget, 1990].

Bibliography

[Clinger & Rees, 1991] William Clinger, Jonathan Rees, *Macros That Work*, Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, January 1991, pp 155–162.

[Curtis & Rauen, 1990] Pavel Curtis, James Rauen, *A Module System for Scheme*, 1990 ACM Conference on Lisp and Functional Programming, Nice, France, June 1990, pp 13-19.

[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.

[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.

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

[Queinnec & Padget, 1990] Christian Queinnec, Julian Padget, *A Model of Modules and Macros for Lisp*, Bath Computing Group Technical Report 90-36, University of Bath, UK.

[Queinnec, 1990] Christian Queinnec, *Compilation of Non-Linear, Second Order Patterns on S-Expressions*, International Workshop PLILP '90, Linköping, Sweden, August 1990, Lecture Notes in Computer Science 456, Springer-Verlag, pp 340–357.

[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

[Steele, 1990] Steele G.L. Jr., *Common Lisp the language*, Second Edition, Digital Press, 1990.

A An example of a macroexpansion module

This module presents the essence of a macroexpansion module. It does not handle errors but just suggests how to define modules implementing the macroexpansion protocol. This module exports a macro, called `defmacro`, to define macros that are coherent with this implementation of syntax. We suppress some details of the exact way in which `lambda` is walked but we give precedence to local variables over macros. This module implements EPS [Dybvig *et al.*, 1988].

```
;;; A simple macroexpansion module

(expose standard)

(immutable
 (union
   (syntax (only (defmacro) (expose macrxpnd)))
   (only (initial-syntax macroexpand-module-body)
     (expose macrxpnd))))

;;; macroexpand: Syntax
;;; keyword: Symbol
;;; expander: Form × Syntax → Form
;;; Build a new syntax similar to macroexpand but where forms whose car
;;; is equal to keyword are expanded thanks to expander.

(set! macro-extend-on-form
  (lambda (macroexpand keyword expander)
    (lambda (e m)
      (if (pair? e)
          (if (eq? (car e) keyword)
              (expander e m)
```

```
              (macroexpand e m))
          (macroexpand e m)))))

(set! macro-walk (lambda (e m) (m e m)))
(set! module-body-macro-walk macro-walk)

(set! macro-walk*
  (lambda (e m)
    (if (pair? e)
        (cons (macro-walk (car e) m)
              (macro-walk* (cdr e) m))
        e)))

(set! initial-syntax
  (lambda (e m)
    (if (pair? e)
        (case (car e)
          ((quote) e)
          ((set!)
           `(set! . ,(macro-walk* (cdr e) m)))
          ((if)
           `(if . ,(macro-walk* (cdr e) m)))
          ((lambda)
           `(lambda
              ,(cadr e)
              . ,(macro-walk* (cddr e) m)))
          ((access) e)
          (else (macro-walk* e m)) )
        e)))

;;; The defmacro form:
;;; syntax: (defmacro name variables . body)
;;; defmacro is itself a macro !

(set! defmacro
  (lambda (s kw modname)
    (macro-extend-on-form s kw
      (lambda (e m)
        (if (if (pair? (cdr e))
              (if (symbol? (car (cdr e)))
                (if (pair? (cdr (cdr e)))
                 (pair? (cdr (cdr (cdr e))))
                   #f) #f) #f)
            (m
              `(set!
                ,(car (cdr e))
                ((lambda (expander)
                   (lambda (syntax
                            keyword
                            module-name)
                     ((access macrxpnd
                              macro-extend-on-form)
                      syntax
                      keyword
                      (lambda (expression
                               macroexpand)
                        (macroexpand
                          ((access primitives apply)
                           expander
                           ((access primitives cdr)
                            expression))
```

```
                macroexpand)))))
         (lambda . ,(cdr (cdr e)))))
   m)
 (error 'defmacro))))))
```