# First-class Global Environments in Common Lisp

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

## ABSTRACT

*Environments* are mentioned in many places in the Common Lisp standard, but the nature of such objects is not specified. For the purpose of this paper, an environment is a mapping from *names* to *meanings*. In a typical Common Lisp implementation the *global environment* is not a first-class object.

In this paper, we advocate first-class global environments, not as an extension or a modification of the Common Lisp standard, but as an implementation technique. We state several advantages in terms of bootstrapping, sandboxing, and more. We show an implementation where there is no performance penalty associated with making the environment first class. For performance purposes, the essence of the implementation relies on the environment containing *cells* (ordinary `cons` cells in our implementation) holding bindings of names to functions and global values that are likely to be heavily solicited at runtime.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Run-time environments*

## General Terms

Algorithms, Languages

## Keywords

CLOS, Common Lisp, Environment

## 1. INTRODUCTION

The Common Lisp standard contains many references to environments. Most of these references concern *lexical* environments at *compile time*, because they are needed in order to process forms in non-null lexical environments. The standard does not specify the nature of these objects, though in CLtL2 [5] there is a suggested protocol that is sometimes supplied in existing Common Lisp implementations.



When it comes to *global environments*, however, the standard is even more reticent. In section 3.2.1 (entitled Compiler Terminology) of the Common Lisp HyperSpec, the distinction is made between the *startup environment*, the *compilation environment*, the *evaluation environment*, and the *runtime environment*. Excluding the runtime environment, the standard allows for all the others to be identical.

In a typical Common Lisp implementation, these global environments are not first-class objects, and there is typically only one global environment available as specifically allowed by the standard. In some implementations, part of the environment is contained in other objects. For instance, it is common for a symbol object to contain a *value cell* containing the global value (if any) of the variable having the name of the symbol and/or a *function cell* containing the definition of a global function having the name of the symbol. This kind of representation is even implicitly suggested by the standard in that it sometimes uses terminology such as *value cell* and *function cell*[1], while pointing out that this terminology is "traditional".

In this paper, we argue that there are many advantages to making the global environment a first-class object. An immediate advantage is that it is then possible to distinguish between the *startup environment*, the *compilation environment* and the *evaluation environment* so that compile-time evaluations by the compiler are not visible in the startup-environment. However, as we show in this paper, there are many more advantages, such as making it easier to create a so-called *sandbox* environment, which is notoriously difficult to do in a typical Common Lisp implementation. Another significant advantage of first-class global environments is that it becomes unnecessary to use temporary package names for bootstrapping a target Common Lisp system from a host Common Lisp system.

In order for first-class global environments to be a viable alternative to the traditional implementation method, they must not incur any performance penalty, at least not at runtime. We show an implementation of first-class global environments that respects this constraint by supplying *cells* that can be thought of as the same as the traditional value cells and function cells, except that they are dislocated so that the are physically located in the environment object as opposed to being associated with a symbol.

[1]See for instance the glossary entries for *cell*, *value cell*, and *function cell* in the HyperSpec.

# 2. PREVIOUS WORK

## 2.1 Gelernter et al

The idea of first-class environments is of course not new. Gelernter et al [1] defined a language called "Symmetric Lisp" in which the programmer is allowed to evaluate expressions with respect to a particular first-class environment. They suggest using this kind of environment as a replacement for a variety of constructs, including closures, structures, classes, and modules. The present paper does not have this kind of ambitious objective, simply because we do not know how to obtain excellent performance for all these constructs with our suggested protocol.

## 2.2 Miller and Rozas

Miller and Rozas [2] describe a set of extensions for the Scheme programming language. In their paper, Miller and Rozas also claim that their proposed first-class environments could serve as a basis for an object-oriented system. Like the present work, Miller and Rozas are concerned with performance, and a large part of their paper is dedicated to this aspect of their proposal.

In their paper, they show that the extensions incurs no performance penalty for code that does not use it. However, for code that uses the extension, the compiler defers accesses to the first-class environment to the *interpreter*, thereby imposing a performance penalty in code using the extension.

The basic mechanism of their proposed extension is a *special form* named `make-environment` that creates and returns an environment in which the code in the *body* of that special form is executed. The operators `lexical-reference` and `lexical-assignment` are provided to access bindings in a first-class environment.

From the examples in the paper, it is clear that their first-class environments are meant to be used at a much finer level of granularity then ours.

## 2.3 Queinnec and Roure

Queinnec and de Roure [3] proposed a protocol for first-class environments in the context of the Scheme programming language. Their motivation is different from ours, in that their environments are meant to be part of the user-visible interface so as to simplify sharing of various objects. However, like the present work, they are also concerned with performance, and they show how to implement their protocol without serious performance degradation.

The environments proposed by Queinnec and de Roure were clearly meant to allow for *modules* as collections of *bindings* where the bindings can be shared by other modules through the use of the new operators `export` and `import`. In contrast, the first-class environments proposed in the present paper are not meant to allow such sharing of bindings, though our proposal might allow such sharing for function and variable bindings.

The paper by Queinnec and de Roure contains a thorough survey of other work related to first-class environments that will not be repeated here.

## 2.4 Garret's lexicons

Ron Garret desicribes *lexicons*[2] which are said to be first-class global environments. However, the concept of a lexicon is very different from the concept of a first-class global environment as defined in this paper.

For one thing, a lexicon is "a mapping from symbols to bindings", which excludes a per-environment set of packages, simply because package names are not symbols, but strings.

Furthermore, a clearly stated goal of lexicons is to create a different Lisp dialect targeted to new users or to users that have prior experience with languages that are different from Lisp.

One explicit goal of lexicons is to replace Common Lisp packages, so that there is a single system-wide symbol with a particular name. In contrast, the first-class environments presented in this paper do not in any way affect the package system. It should be noted that with our first-class environments, *symbols are still unique*, i.e., for a given package $P$ and a given symbol name $N$, there is at most one symbol with the name $N$ in $P$; independently of the number of first-class environments in the system.

Garret discusses the use of lexicons as modules and shows examples where functions defined in one lexicon can be imported into a different lexicon. The use of the operator `use-lexicon` imports all the *bindings* of an explicitly-mentioned lexicon into the current one. In the present work, we do not emphasize the possibility of sharing bindings between first-class environment. However, since functions and global values of special variables are stored in indirections called *cells* in our environment, such sharing of bindings would also be possible in the first-class environments presented in this paper.

# 3. OUR TECHNIQUE

We suggest a CLOS-based *protocol* defining the set of operations on a first-class environment. This protocol contains around 40 generic functions.

Mainly, the protocol contains versions of Common Lisp environment functions such as `fboundp`, `find-class`, etc. that take an additional required `environment` argument.

In addition to these functions, the protocol contains a set of functions for accessing *cells* that in most implementations would be stored elsewhere. Thus, a binding of a function name to a function object contains an indirection in the form of a *function cell*. The same holds for the binding of a variable name (a symbol) to its *global value*. In our implementation, these cells are ordinary `cons` cells with the `car` containing the value of the binding, and the `cdr` containing `nil`.

These cells are created as needed. The first time a reference to a function is made, the corresponding cell is created. Compiled code that refers to a global function will have the

[2]Unpublished document. A PDF version can be found here: http://www.flownet.com/ron/lisp/lexicons.pdf.

corresponding cell in its run-time environment. The cost of accessing a function at run-time is therefore no greater in our implementation than in an implementation that accesses the function through the symbol naming it.

Our technique does, however, incur a performance penalty for functions such as `fdefinition` and `symbol-value` with an argument that is computed at run-time[3] compared to an implementation in which each symbol contains slots for these objects. However, even in a high-performance implementation such as SBCL, these values are *not* contained in symbol slots.

# 4. BENEFITS OF OUR METHOD

## 4.1 Native compilation

The Common Lisp standards suggests that the *startup environment* and the *evaluation environment* may be different.[4] Our method allows most evaluations by the compiler to have no influence in the startup environment. It suffices to *clone* the startup environment in order to obtain the evaluation environment.

With the tradition of the startup environment and evaluation environment being identical, some evaluations by the compiler would have side effects in the startup environment. In particular, the value cells and function cells are shared. Therefore, executing code at compile time that alters the global binding of a function or a variable will also be seen in the startup environment.

As an example of code that should not be evaluated in the startup environment, consider definitions of macros that are only required for the correct compilation of some program, as well as definitions of functions that are only required for the expansion of such macros. A definition for this purpose might be wrapped in an `eval-when` form with `:compile-toplevel` as the only *situation* in which the definition should be evaluated. When the evaluation environment and the startup environment are identical, such a definition will be evaluated in the startup environment, and persist after the program has been compiled.

## 4.2 Bootstrapping

For the purpose of this paper, we use the word *bootstrapping* to mean the process of building the executable of some implementation (the *target* system) by executing code in the running process of another implementation (the *host* system). The host and the target systems may be the same implementation. In this context, a *cross compiler* is a compiler that executes in the host system while generating code for the target system.

When a *host* Common Lisp system is used to bootstrap a *target* Common Lisp system, the target system needs its own definitions of many standard Common Lisp features. In particular, in order to compile code for the target system in the host system, the cross compiler needs access to the target definitions of standard Common Lisp macros, in particular the defining macros such as `defun`, `defmacro`, `defgeneric`, `defvar`, etc.

It is, of course, not an option to replace the host versions of such macros with the corresponding target versions. Doing so would almost certainly break the host system in irreparable ways. To avoid that the system might be damaged this way, many Common Lisp systems have a feature called *package locks*[5] which prevents the redefinition of standard Common Lisp functions, macros, etc.

To deal with the problem of bootstrapping, some systems, in particular SBCL, replace the standard package names by some other names for target code, typically derived from the standard names in some systematic way [4]. Using different package names guarantees that there is no clash between a host package name and the corresponding target package name. However, using non-standard package names also means that the text of the source code for the target will not correspond to the target code that ends up in the final system.

As an alternative to renaming packages, first-class global environments represent an elegant solution to the bootstrapping problem. In a system that already supports first-class global environments, creating a new such environment in which the target definitions are allowed to replace standard Common Lisp definitions is of course very simple. But even in a host system that does not a priori support first-class global environments, it is not very difficult to create such environments.

Making the cross compiler access such a first-class global environment is just a matter of structuring its environment-lookup functions so that they do not directly use standard Common Lisp functions such as `fboundp` or `fdefinition`, and instead use the generic functions of the first-class global environment protocol.

## 4.3 Sandboxing

It is notoriously hard to create a so-called *sandbox environment* for Common Lisp, i.e., an environment that contains a "safe" subset of the full language. A typical use case would be to provide a Read-Eval-Print Loop accessible through a web interface for educational purposes. Such a sandbox environment is hard to achieve because functions such as `eval` and `compile` would have to be removed so that the environment could not be destroyed by a careless user. However, these functions are typically used by parts of the system. For example, CLOS might need the compiler in order to generate dispatch code.

The root of the problem is that in Common Lisp there is always a way for the user of a Read-Eval-Print Loop to access

[3]When the argument is a constant, a suitable *compiler-macro* can turn the form into an access of the corresponding cell.

[4]Recall that the startup environment is the global environment as it was when the compilation was initiated, and that the evaluation environment is the global environment in which evaluations initiated by the compiler are accomplished.

[5]The name of this feature is misleading. While it does make sure that the protected package is not modified, it also makes sure that functions, macros, etc., with names in the package are not redefined. Such redefinitions do not alter the package itself, of course.

every global function in the system, including the compiler. While it might be easy to remove functions that may render the system unusable *directly* such as functions for opening and deleting files, it is generally not possible to remove the compiler, since it is used at run-time to evaluate expressions and in many systems in order to create functions for generic dispatch. With access to the compiler, a user can potentially create and execute code for any purpose.

Using first-class global environments solves this problem in an elegant way. It suffices to provide a restricted environment in which there is no binding from the names `eval` and `compile` to the corresponding functions. These functions can still be available in some other environment for use by the system itself.

### 4.4 Multiple package versions

When running multiple applications in the same Common Lisp process, there can easily be conflicts between different versions of the same package. First-class global environments can alleviate this problem by having different global environments for the different applications causing the conflict.

Suppose, for instance, that applications $A$ and $B$ both require some Common Lisp package $P$, but that $P$ exists in different *versions*. Suppose also that $A$ and $B$ require different such versions of $P$. Since the Common Lisp standard has no provisions for multiple versions of a package, it becomes difficult to provide both $A$ and $B$ in the same Common Lisp process.

Using first-class global environments as proposed in this paper, two different global environments can be created for building $A$ and $B$. These two environments would differ in that the name $P$ would refer to different versions of the package $P$.

### 4.5 Separate environment for each application

Taking the idea of Section 4.4 even further, it is sometimes desirable for a large application to use a large number of packages that are specific to that application. In such a situation, it is advantageous to build the application in a separate global environment, so that the application-specific packages exist only in that environment. The main entry point(s) of the application can then be made available in other environments without making its packages available.

Using separate first-class global environments for this purpose would also eliminate the problem of choosing package names for an application that are guaranteed not to conflict with names of packages in other applications that some user might simultaneously want to install.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we have advocated first-class global environments as a way of implementing the global environments mentioned in the HyperSpec. We have seen that this technique has several advantages in terms of flexibility of the system, and that it greatly simplifies certain difficult problems such as bootstrapping and sandboxing.

An interesting extension of our technique would be to consider *environment inheritance*[6] For example, an environment providing the standard bindings of the Common Lisp language could be divided into an immutable part and a mutable part. The mutable part would then contain features that can be modified by the user, such as the generic function `print-object` or the variable `*print-base*`, and it would inherit from the immutable part. With this feature, it would only be necessary to clone the mutable part in order to create the evaluation environment from the startup environment as suggested in Section 4.1.

We also think that first-class global environments could be an excellent basis for a multi-user Common Lisp system.

In such a system, each user would have an initial, private, environment. That environment would contain the standard Common Lisp functionality. Most standard Common Lisp functions would be shared between all users. Some functions, such as `print-object` or `initialize-instance` would not be shared, so as to allow individual users to add methods to them without affecting other users.

Furthermore, functionality that could destroy the integrity of the system, such as access to raw memory, would be accessible in an environment reserved for system maintenance. This environment would not be accessible to ordinary users.

## 6. ACKNOWLEDGMENTS

We would like to thank Alastair Bridgewater, David Murray, Robert Smith, Nicolas Hafner, and Bart Botta for providing valuable feedback on early versions of this paper.

## APPENDIX

## A. PROTOCOL

In this appendix we present the generic functions making up the protocol for our first-class global environments. The definitions here should be considered *preliminary*, because there are some aspects of this protocol that need further consideration. As an example, consider the function `function-lambda-list`. We have not made up our minds as to whether this function should be part of the protocol, or just a function to be applied to function objects.

`fboundp` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fboundp`.

It returns true if *fname* has a definition in *env* as an ordinary function, a generic function, a macro, or a special operator.

`fmakunbound` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fmakunbound`.

Makes *fname* unbound in the function namespace of *env*.

If *fname* already has a definition in *env* as an ordinary func-

[6]We mean *inheritance* not in the sense of subclassing, but rather as used in section 3.2.1 of the HyperSpec.

tion, as a generic function, as a macro, or as a special operator, then that definition is lost.

If *fname* has a `setf` expander associated with it, then that `setf` expander is lost.

`special-operator` *fname env* [*GF*]

If *fname* has a definition as a special operator in *env*, then that definition is returned. The definition is the object that was used as an argument to `(setf special-operator)`. The exact nature of this object is not specified, other than that it can not be `nil`. If *fname* does not have a definition as a special operator in *env*, then `nil` is returned.

`(setf special-operator)` *new-def fname env* [*GF*]

Set the definition of *fname* to be a special operator. The exact nature of *new-def* is not specified, except that a value of `nil` means that *fname* no longer has a definition as a special operator in *env*.

If a value other than `nil` is given for *new-def*, and *fname* already has a definition as an ordinary function, as a generic function, or as a macro, then an error is signaled. As a consequence, if it is desirable for *fname* to have a definition both as a special operator and as a macro, then the definition as a special operator should be set first.

`fdefinition` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fdefinition`.

If *fname* has a definition in the function namespace of *env* (i.e., if `fboundp` returns true), then a call to this function succeeds. Otherwise an error of type `undefined-function` is signaled.

If *fname* is defined as an ordinary function or a generic function, then a call to this function returns the associated function object.

If *fname* is defined as a macro, then a list of the form `(cl:macro-function` *function*`)` is returned, where *function* is the macro expansion function associated with the macro.

If *fname* is defined as a special operator, then a list of the form `(cl:special` *object*`)` is returned, where the nature of *object* is currently not specified.

`(setf fdefinition)` *new-def fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fdefinition`.

*new-def* must be an ordinary function or a generic function. If *fname* already names a function or a macro, then the previous definition is lost. If *fname* already names a special operator, then an error is signaled.

If *fname* is a symbol and it has an associated `setf` expander, then that `setf` expander is preserved.

`macro-function` *symbol env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:macro-function`.

If *symbol* has a definition as a macro in *env*, then the corresponding macro expansion function is returned.

If *symbol* has no definition in the function namespace of *env*, or if the definition is not a macro, then this function returns `nil`.

`(setf macro-function)` *new-def symbol env* [*GF*]

This generic function is a generic version of the Common Lisp function `(setf cl:macro-function)`.

*new-def* must be a macro expansion function or `nil`. A call to this function then always succeeds. A value of `nil` means that the *symbol* no longer has a macro function associated with it. If *symbol* already names a macro or a function, then the previous definition is lost. If *symbol* already names a special operator, that definition is kept.

If *symbol* already names a function, then any proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed type will be `t`.

If *symbol* already names a function, then any `inline` or `notinline` proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed inline information will be `nil`.

If *fname* is a symbol and it has an associated `setf` expander, then that `setf` expander is preserved.

`compiler-macro-function` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:compiler-macro-function`.

If *fname* has a definition as a compiler macro in *env*, then the corresponding compiler macro function is returned.

If *fname* has no definition as a compiler macro in *env*, then this function returns `nil`.

`(setf compiler-macro-function)` *new-def fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `(setf cl:compiler-macro-function)`.

*new-def* can be a compiler macro function or `nil`. When it is a compiler macro function, then it establishes *new-def* as a compiler macro for *fname* and any existing definition is lost. A value of `nil` means that *fname* no longer has a compiler macro associated with it in *env*.

`function-type` *fname env* [*GF*]

This generic function returns the proclaimed type of the function associated with *fname* in *env*.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

If *fname* is associated with an ordinary function or a generic function in *env*, but no type proclamation for that function has been made, then this generic function returns `t`.

`(setf function-type)` *new-type fname env* [*GF*]

This generic function is used to set the proclaimed type of the function associated with *fname* in *env* to *new-type*.

If *fname* is associated with a macro or a special operator in *env*, then an error is signaled.

`function-inline` *fname env* [*GF*]

This generic function returns the proclaimed inline information of the function associated with *fname* in *env*.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

If *fname* is associated with an ordinary function or a generic function in *env*, then the return value of this function is either `nil`, `inline`, or `notinline`. If no inline proclamation has been made, then this generic function returns `nil`.

`(setf function-inline)` *new-inline fname env* [*GF*]

This generic function is used to set the proclaimed inline information of the function associated with *fname* in *env* to *new-inline*.

*new-inline* must have one of the values `nil`, `inline`, or `notinline`.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

`function-cell` *fname env* [*GF*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the function named *fname*. When *fname* has no definition as a function, the `car` of this cell will contain a function that, when called, signals an error of type `undefined-function`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment.

`function-unbound` *fname env* [*GF*]

A call to this function always succeeds. It returns a function that, when called, signals an error of type `undefined-function`. When *fname* has no definition as a function, the return value of this function is the contents of the `cons` cell returned by `function-cell`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment. Client code can use the return value of this function to determine whether *fname* is unbound and if so signal an error when an attempt is made to evaluate the form `(function` *fname*`)`.

`function-lambda-list` *fname env* [*GF*]

This function returns two values. The first value is an ordinary lambda list, or `nil` if no lambda list has been defined for *fname*. The second value is true if and only if a lambda list has been defined for *fname*.

`boundp` *symbol env* [*GF*]

It returns true if *symbol* has a definition in *env* as a constant variable, as a special variable, or as a symbol macro. Otherwise, it returns `nil`.

`constant-variable` *symbol env* [*GF*]

This function returns the value of the constant variable *symbol*.

If *symbol* does not have a definition as a constant variable, then an error is signaled.

`(setf constant-variable)` *value symbol env* [*GF*]

This function is used in order to define *symbol* as a constant variable in *env*, with *value* as its value.

If *symbol* already has a definition as a special variable or as a symbol macro in *env*, then an error is signaled.

If *symbol* already has a definition as a constant variable, and its current value is not `eql` to *value*, then an error is signaled.

`special-variable` *symbol env* [*GF*]

This function returns two values. The first value is the value of *symbol* as a special variable in *env*, or `nil` if *symbol* does not have a value as a special variable in *env*. The second value is true if *symbol* does have a value as a special variable in *env* and `nil` otherwise.

Notice that the symbol can have a value even though this function returns `nil` and `nil`. The first such case is when the symbol has a value as a constant variable in *env*. The second case is when the symbol was assigned a value using `(setf symbol-value)` without declaring the variable as `special`.

`(setf special-variable)` *value symbol env init-p* [*GF*]

This function is used in order to define *symbol* as a special variable in *env*.

If *symbol* already has a definition as a constant variable or as a symbol macro in *env*, then an error is signaled. Otherwise, *symbol* is defined as a special variable in *env*.

If *symbol* already has a definition as a special variable, and *init-p* is `nil`, then this function has no effect. The current value is not altered, or if *symbol* is currently unbound, then it remains unbound.

If *init-p* is true, then *value* becomes the new value of the special variable *symbol*.

`symbol-macro` *symbol env* [*GF*]

This function returns two values. The first value is a macro expansion function associated with the symbol macro named by *symbol*, or `nil` if *symbol* does not have a definition as a symbol macro. The second value is the form that *symbol* expands to as a macro, or `nil` if symbol does not have a definition as a symbol macro.

It is guaranteed that the same (in the sense of `eq`) function is returned by two consecutive calls to this function with the same symbol as the first argument, as long as the definition of *symbol* does not change.

`(setf symbol-macro)` *expansion symbol env* [*GF*]

This function is used in order to define *symbol* as a symbol macro with the given *expansion* in *env*.

If *symbol* already has a definition as a constant variable, or as a special variable, then an error of type `program-error` is signaled.

`variable-type` *symbol env* [*GF*]

This generic function returns the proclaimed type of the variable associated with *symbol* in *env*.

If *symbol* has a definition as a constant variable in *env*, then the result of calling `type-of` on its value is returned.

If *symbol* does not have a definition as a constant variable in *env*, and no previous type proclamation has been made for *symbol* in *env*, then this function returns `t`.

`(setf variable-type)` *new-type symbol env* [*GF*]

This generic function is used to set the proclaimed type of the variable associated with *symbol* in *env*.

If *symbol* has a definition as a constant variable in *env*, then an error is signaled.

It is meaningful to set the proclaimed type even if *symbol* has not previously been defined as a special variable or as a symbol macro, because it is meaningful to use `(setf symbol-value)` on such a symbol.

Recall that the HyperSpec defines the meaning of proclaiming the type of a symbol macro. Therefore, it is meaningful to call this function when *symbol* has a definition as a symbol macro in *env*.

`variable-cell` *symbol env* [*GF*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the variable named *symbol*. When *symbol* has no definition as a variable, the `car` of this cell will contain an object that indicates that the variable is unbound. This object is the return value of the function `variable-unbound`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment.

`variable-unbound` *symbol env* [*GF*]

A call to this function always succeeds. It returns an object that indicates that the variable is unbound. The `cons` cell returned by the function `variable-cell` contains this object whenever the variable named *symbol* is unbound. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment. Client code can use the return value of this function to determine whether *symbol* is unbound.

`find-class` *symbol env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:find-class`.

If *symbol* has a definition as a class in *env*, then that class metaobject is returned. Otherwise `nil` is returned.

`(setf find-class)` *new-class symbol env* [*GF*]

This generic function is a generic version of the Common Lisp function `(setf cl:find-class)`.

This function is used in order to associate a class with a class name in *env*.

If *new-class* is a class metaobject, then that class metaobject is associated with the name *symbol* in *env*. If *symbol* already names a class in *env* than that association is lost.

If *new-class* is `nil`, then *symbol* is no longer associated with a class in *env*.

If *new-class* is neither a class metaobject nor `nil`, then an error of type `type-error` is signaled.

`setf-expander` *symbol env* [*GF*]

This generic function returns the `setf` expander associated with *symbol* in *env*. If *symbol* is not associated with any `setf` expander in *env*, then `nil` is returned.

`(setf setf-expander)` *new-expander symbol env* [*GF*]

This generic function is used to set the `setf` expander associated with *symbol* in *env*.

If *symbol* is not associated with an ordinary function, a generic function, or a macro in *env*, then an error is signaled.

If there is already a `setf` expander associated with *symbol* in *env*, then the old `setf` expander is lost.

If a value of `nil` is given for *new-expander*, then any current `setf` expander associated with *symbol* is removed. In this case, no error is signaled, even if *symbol* is not associated with any ordinary function, generic function, or macro in *env*.

`default-setf-expander` *env* [*GF*]

This generic function returns the default `setf` expander, to

be used when the function `setf-expander` returns `nil`. This function always returns a valid `setf` expander.

`(setf default-setf-expander)` *new-expander env* [*GF*]

This generic function is used to set the default `setf` expander in `env`.

`type-expander` *symbol env* [*GF*]

This generic function returns the type expander associated with *symbol* in `env`. If *symbol* is not associated with any type expander in `env`, then `nil` is returned.

`(setf type-expander)` *new-expander symbol env* [*GF*]

This generic function is used to set the type expander associated with *symbol* in `env`.

If there is already a type expander associated with *symbol* in `env`, then the old type expander is lost.

`find-package` *name env* [*GF*]

Return the package with the name or the nickname *name* in the environment *env*. If there is no package with that name in *env*, then return `nil`. Contrary to the standard Common Lisp function `cl:find-package`, for this function, *name* must be a string.

`package-name` *package env* [*GF*]

Return the string that names *package* in *env*. If *package* is not associated with any name in *env*, then `nil` is returned. Contrary to the standard Common Lisp function `cl:package-name`, for this function, *package* must be a package object.

`(setf package-name)` *new-name package env* [*GF*]

Make the string *new-name* the new name of *package* in *env*. If *new-name* is `nil`, then *package* no longer has a name in *env*.

`package-nicknames` *package env* [*GF*]

Return a list of the strings that are nicknames of *package* in *env*. Contrary to the standard Common Lisp function `cl:package-nicknames`, for this function, *package* must be a package object.

`(setf package-nicknames)` *new-names package env* [*GF*]

Associate the strings in the list *new-names* as nicknames of *package* in *env*.

# B. REFERENCES


[1] D. Gelernter, S. Jagannathan, and T. London. Environments as first class objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 98–110, New York, NY, USA, 1987. ACM.

[2] J. S. Miller and G. J. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation*, 4(2):107–141, Mar. 1991.

[3] C. Queinnec and D. de Roure. Sharing code through first-class environments. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 251–261, New York, NY, USA, 1996. ACM.

[4] C. Rhodes. Self-sustaining systems. chapter SBCL: A Sanely-Bootstrappable Common Lisp, pages 74–86. Springer-Verlag, Berlin, Heidelberg, 2008.

[5] G. L. Steele, Jr. *Common LISP: The Language (2nd Ed.)*. Digital Press, Newton, MA, USA, 1990.