# Chapter 11

# Packages

One problem with earlier Lisp systems is the use of a single name space for all symbols. In large Lisp systems, with modules written by many different programmers, accidental name collisions become a serious problem. Common Lisp addresses this problem through the *package system*, derived from an earlier package system developed for Lisp Machine Lisp [55]. In addition to preventing name-space conflicts, the package system makes the modular structure of large Lisp systems more explicit.

A *package* is a data structure that establishes a mapping from print names (strings) to symbols. The package thus replaces the "oblist" or "obarray" machinery of earlier Lisp systems. At any given time one package is current, and this package is used by the Lisp reader in translating strings into symbols. The current package is, by definition, the one that is the value of the global variable ***package***. It is possible to refer to symbols in packages other than the current one through the use of *package qualifiers* in the printed representation of the symbol. For example, **foo:bar**, when seen by the reader, refers to the symbol whose name is **bar** in the package whose name is **foo**. (Actually, this is true only if **bar** is an external symbol of **foo**, that is, a symbol that is supposed to be visible outside of **foo**. A reference to an internal symbol requires the intentionally clumsier syntax **foo::bar**.)

The string-to-symbol mappings available in a given package are divided into two classes, *external* and *internal*. We refer to the symbols accessible via these mappings as being *external* and *internal* symbols of the package in question, though really it is the mappings that are different and not the symbols themselves. Within a given package, a name refers to one symbol or to none; if it does refer to a symbol, then it is either external or internal

in that package, but not both.

External symbols are part of the package's public interface to other packages. External symbols are supposed to be chosen with some care and are advertised to users of the package. Internal symbols are for internal use only, and these symbols are normally hidden from other packages. Most symbols are created as internal symbols; they become external only if they appear explicitly in an **export** command for the package.

A symbol may appear in many packages. It will always have the same name wherever it appears, but it may be external in some packages and internal in others. On the other hand, the same name (string) may refer to different symbols in different packages.

Normally, a symbol that appears in one or more packages will be *owned* by one particular package, called the *home package* of the symbol; that package is said to *own* the symbol. Every symbol has a component called the *package cell* that contains a pointer to its home package. A symbol that is owned by some package is said to be *interned*. Some symbols are not owned by any package; such a symbol is said to be *uninterned*, and its package cell contains **nil**.

Packages may be built up in layers. From the point of view of a package's user, the package is a single collection of mappings from strings into internal and external symbols. However, some of these mappings may be established within the package itself, while other mappings are inherited from other packages via the **use-package** construct. (The mechanisms responsible for this inheritance are described below.) In what follows, we will refer to a symbol as being *accessible* in a package if it can be referred to without a package qualifier when that package is current, regardless of whether the mapping occurs within that package or via inheritance. We will refer to a symbol as being *present* in a package if the mapping is in the package itself and is not inherited from somewhere else. Thus a symbol present in a package is accessible, but an accessible symbol is not necessarily present.

A symbol is said to be *interned in a package* if it is accessible in that package and also is owned (by either that package or some other package). Normally all the symbols accessible in a package will in fact be owned by some package, but the terminology is useful when discussing the pathological case of an accessible but unowned (uninterned) symbol.

As a verb, to *intern* a symbol in a package means to cause the symbol to be interned in the package if it was not already; this process is performed by the function **intern**. If the symbol was previously unowned, then the package

it is being interned in becomes its owner (home package); but if the symbol was previously owned by another package, that other package continues to own the symbol.

To *unintern* a symbol from the package means to cause it to be not present in the package and, additionally, to cause the symbol to be uninterned if the package was the home package (owner) of the symbol. This process is performed by the function **unintern**.

## 11.1 Consistency Rules

Package-related bugs can be very subtle and confusing: things are not what they appear to be. The Common Lisp package system is designed with a number of safety features to prevent most of the common bugs that would otherwise occur in normal use. This may seem over-protective, but experience with earlier package systems has shown that such safety features are needed.

In dealing with the package system, it is useful to keep in mind the following consistency rules, which remain in force as long as the value of ***package*** is not changed by the user:

- *Read-read consistency:* Reading the same print name always results in the same (**eq**) symbol.
- *Print-read consistency:* An interned symbol always prints as a sequence of characters that, when read back in, yields the same (**eq**) symbol.
- *Print-print consistency:* If two interned symbols are not **eq**, then their printed representations will be different sequences of characters.

These consistency rules remain true in spite of any amount of implicit interning caused by typing in Lisp forms, loading files, etc. This has the important implication that, as long as the current package is not changed, results are reproducible regardless of the order of loading files or the exact history of what symbols were typed in when. The rules can only be violated by explicit action: changing the value of ***package***, forcing some action by continuing from an error, or calling one of the "dangerous" functions **unintern**, **unexport**, **shadow**, **shadowing-import**, or **unuse-package**.

## 11.2 Package Names

Each package has a name (a string) and perhaps some nicknames. These are assigned when the package is created, though they can be changed later. A package's name should be something long and self-explanatory, like **editor**; there might be a nickname that is shorter and easier to type, such as **ed**.

There is a single name space for packages. The function **find-package** translates a package name or nickname into the associated package. The function **package-name** returns the name of a package. The function **package-nicknames** returns a list of all nicknames for a package. The function **rename-package** removes a package's current name and nicknames and replaces them with new ones specified by the user. Package renaming is occasionally useful when, for development purposes, it is desirable to load two versions of a package into the same Lisp. One can load the first version, rename it, and then load the other version, without getting a lot of name conflicts.

When the Lisp reader sees a qualified symbol, it handles the package-name part in the same way as the symbol part with respect to capitalization. Lowercase characters in the package name are converted to corresponding uppercase characters unless preceded by the escape character \ or surrounded by | characters. The lookup done by the **find-package** function is case-sensitive, like that done for symbols. Note that **|Foo|:|Bar|** refers to a symbol whose name is **Bar** in a package whose name is **Foo**. By contrast, **|Foo:Bar|** refers to a seven-character symbol that has a colon in its name (as well as two uppercase letters and four lowercase letters) and is interned in the current package. Following the convention used in this book for symbols, we show ordinary package names using lowercase letters, even though the name string is internally represented with uppercase letters.

Most of the functions that require a package-name argument from the user accept either a symbol or a string. If a symbol is supplied, its print name will be used; the print name will already have undergone case-conversion by the usual rules. If a string is supplied, it must be so capitalized as to match exactly the string that names the package.

X3J13 voted in January 1989 to clarify that one may use either a package object or a package name (symbol or string) in any of the following situations:

- the **:use** argument to **make-package**
- the first argument to **package-use-list**, **package-used-by-list**,

**package-name**, **package-nicknames**, **in-package**, **find-package**, **rename-package**, or **delete-package**,

- the second argument to **intern**, **find-symbol**, **unintern**, **export**, **unexport**, **import**, **shadowing-import**, or **shadow**
- the first argument, or a member of the list that is the first argument, to **use-package** or **unuse-package**
- the value of the *package* given to **do-symbols**, **do-external-symbols**, or **do-all-symbols**
- a member of the *package-list* given to **with-package-iterator**

Note that the first argument to **make-package** must still be a package name and not an actual package; it makes no sense to create an already existing package. Similarly, package nicknames must always be expressed as package names and not as package objects. If **find-package** is given a package object instead of a name, it simply returns that package.

## 11.3 Translating Strings to Symbols

The value of the special variable ***package*** must always be a package object (not a name). Whatever package object is currently the value of ***package*** is referred to as the *current package*.

When the Lisp reader has, by parsing, obtained a string of characters thought to name a symbol, that name is looked up in the current package. This lookup may involve looking in other packages whose external symbols are inherited by the current package. If the name is found, the corresponding symbol is returned. If the name is not found (that is, there is no symbol of that name accessible in the current package), a new symbol is created for it and is placed in the current package as an internal symbol. Moreover, the current package becomes the owner (home package) of the symbol, and so the symbol becomes interned in the current package. If the name is later read again while this same package is current, the same symbol will then be found and returned.

Often it is desirable to refer to an external symbol in some package other than the current one. This is done through the use of a *qualified name*, consisting of a package name, then a colon, then the name of the symbol.

This causes the symbol's name to be looked up in the specified package, rather than in the current one. For example, **editor:buffer** refers to the external symbol named **buffer** accessible in the package named **editor**, regardless of whether there is a symbol named **buffer** in the current package. If there is no package named **editor**, or if no symbol named **buffer** is accessible in **editor**, or if **buffer** is an internal symbol in **editor**, the Lisp reader will signal a correctable error to ask the user for instructions.

On rare occasions, a user may need to refer to an *internal* symbol of some package other than the current one. It is illegal to do this with the colon qualifier, since accessing an internal symbol of some other package is usually a mistake. However, this operation is legal if a doubled colon **::** is used as the separator in place of the usual single colon. If **editor::buffer** is seen, the effect is exactly the same as reading **buffer** with ***package*** temporarily rebound to the package whose name is **editor**. This special-purpose qualifier should be used with caution.

The package named **keyword** contains all keyword symbols used by the Lisp system itself and by user-written code. Such symbols must be easily accessible from any package, and name conflicts are not an issue because these symbols are used only as labels and never to carry package-specific values or properties. Because keyword symbols are used so frequently, Common Lisp provides a special reader syntax for them. Any symbol preceded by a colon but no package name (for example **:foo**) is added to (or looked up in) the **keyword** package as an *external* symbol. The **keyword** package is also treated specially in that whenever a symbol is added to the **keyword** package the symbol is always made external; the symbol is also automatically declared to be a constant (see **defconstant**) and made to have itself as its value. This is why every keyword evaluates to itself. As a matter of style, keywords should always be accessed using the leading-colon convention; the user should never import or inherit keywords into any other package. It is an error to try to apply **use-package** to the **keyword** package.

Each symbol contains a package cell that is used to record the home package of the symbol, or **nil** if the symbol is uninterned. This cell may be accessed by using the function **symbol-package**. When an interned symbol is printed, if it is a symbol in the keyword package, then it is printed with a preceding colon; otherwise, if it is accessible (directly or by inheritance) in the current package, it is printed without any qualification; otherwise, it is printed with the name of the home package as the qualifier, using **:** as the separator if the symbol is external and **::** if not.

A symbol whose package slot contains **nil** (that is, has no home package) is printed preceded by #:. It is possible, by the use of **import** and **unintern**, to create a symbol that has no recorded home package but that in fact is accessible in some package. The system does not check for this pathological case, and such symbols will always be printed preceded by #:.

In summary, the following four uses of symbol qualifier syntax are defined.

**foo:bar** When read, looks up **BAR** among the external symbols of the package named **FOO**. Printed when symbol **bar** is external in its home package **foo** and is not accessible in the current package.

**foo::bar** When read, interns **BAR** as if **FOO** were the current package. Printed when symbol **bar** is internal in its home package **foo** and is not accessible in the current package.

**:bar** When read, interns **BAR** as an external symbol in the **keyword** package and makes it evaluate to itself. Printed when the home package of symbol **bar** is **keyword**.

**#:bar** When read, creates a new uninterned symbol named **BAR**. Printed when the symbol **bar** is uninterned (has no home package), even in the pathological case that **bar** is uninterned but nevertheless somehow accessible in the current package.

All other uses of colons within names of symbols are not defined by Common Lisp but are reserved for implementation-dependent use; this includes names that end in a colon, contain two or more colons, or consist of just a colon.

## 11.4 Exporting and Importing Symbols

Symbols from one package may be made accessible in another package in two ways.

First, any individual symbol may be added to a package by use of the function **import**. The form **(import 'editor:buffer)** takes the external symbol named **buffer** in the **editor** package (this symbol was located when the form was read by the Lisp reader) and adds it to the current package as an internal symbol. The symbol is then present in the current package. The

imported symbol is not automatically exported from the current package, but if it is already present and external, then the fact that it is external is not changed. After the call to **import** it is possible to refer to **buffer** in the importing package without any qualifier. The status of **buffer** in the package named **editor** is unchanged, and **editor** remains the home package for this symbol. Once imported, a symbol is *present* in the importing package and can be removed only by calling **unintern**.

If the symbol is already present in the importing package, **import** has no effect. If a distinct symbol with the name **buffer** is accessible in the importing package (directly or by inheritance), then a correctable error is signaled, as described in section 11.5, because **import** avoids letting one symbol shadow another.

A symbol is said to be *shadowed* by another symbol in some package if the first symbol would be accessible by inheritance if not for the presence of the second symbol. To import a symbol without the possibility of getting an error because of shadowing, use the function **shadowing-import**. This inserts the symbol into the specified package as an internal symbol, regardless of whether another symbol of the same name will be shadowed by this action. If a different symbol of the same name is already present in the package, that symbol will first be uninterned from the package (see **unintern**). The new symbol is added to the package's shadowing-symbols list. **shadowing-import** should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

The second mechanism is provided by the function **use-package**. This causes a package to inherit all of the external symbols of some other package. These symbols become accessible as *internal* symbols of the using package. That is, they can be referred to without a qualifier while this package is current, but they are not passed along to any other package that uses this package. Note that **use-package**, unlike **import**, does not cause any new symbols to be *present* in the current package but only makes them *accessible* by inheritance. **use-package** checks carefully for name conflicts between the newly imported symbols and those already accessible in the importing package. This is described in detail in section 11.5.

Typically a user, working by default in the **user** package, will load a number of packages into Lisp to provide an augmented working environment, and then call **use-package** on each of these packages to allow easy access to their external symbols. **unuse-package** undoes the effects of a previous **use-package**. The external symbols of the used package are no

longer inherited. However, any symbols that have been imported into the using package continue to be present in that package.

There is no way to inherit the *internal* symbols of another package; to refer to an internal symbol, the user must either make that symbol's home package current, use a qualifier, or import that symbol into the current package.

The distinction between external and internal symbols is a primary means of hiding names so that one program does not tread on the namespace of another.

When **intern** or some other function wants to look up a symbol in a given package, it first looks for the symbol among the external and internal symbols of the package itself; then it looks through the external symbols of the used packages in some unspecified order. The order does not matter; according to the rules for handling name conflicts (see below), if conflicting symbols appear in two or more packages inherited by package $X$, a symbol of this name must also appear in $X$ itself as a shadowing symbol. Of course, implementations are free to choose other, more efficient ways of implementing this search, as long as the user-visible behavior is equivalent to what is described here.

The function **export** takes a symbol that is accessible in some specified package (directly or by inheritance) and makes it an external symbol of that package. If the symbol is already accessible as an external symbol in the package, **export** has no effect. If the symbol is directly present in the package as an internal symbol, it is simply changed to external status. If it is accessible as an internal symbol via **use-package**, the symbol is first imported into the package, then exported. (The symbol is then present in the specified package whether or not the package continues to use the package through which the symbol was originally inherited.) If the symbol is not accessible at all in the specified package, a correctable error is signaled that, upon continuing, asks the user whether the symbol should be imported.

The function **unexport** is provided mainly as a way to undo erroneous calls to **export**. It works only on symbols directly present in the current package, switching them back to internal status. If **unexport** is given a symbol already accessible as an internal symbol in the current package, it does nothing; if it is given a symbol not accessible in the package at all, it signals an error.

## 11.5 Name Conflicts

A fundamental invariant of the package system is that within one package any particular name can refer to at most one symbol. A *name conflict* is said to occur when there is more than one candidate symbol and it is not obvious which one to choose. If the system does not always choose the same way, the read-read consistency rule would be violated. For example, some programs or data might have been read in under a certain mapping of the name to a symbol. If the mapping changes to a different symbol, and subsequently additional programs or data are read, then the two programs will not access the same symbol even though they use the same name. Even if the system did always choose the same way, a name conflict is likely to result in a mapping from names to symbols different from what was expected by the user, causing programs to execute incorrectly. Therefore, any time a name conflict is about to occur, an error is signaled. The user may continue from the error and tell the package system how to resolve the conflict.

It may be that the same symbol is accessible to a package through more than one path. For example, the symbol might be an external symbol of more than one used package, or the symbol might be directly present in a package and also inherited from another package. In such cases there is no name conflict. The same identical symbol cannot conflict with itself. Name conflicts occur only between distinct symbols with the same name.

The creator of a package can tell the system in advance how to resolve a name conflict through the use of *shadowing*. Every package has a list of shadowing symbols. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. A name conflict involving a shadowing symbol is always resolved in favor of the shadowing symbol, without signaling an error (except for one instance involving **import** described below). The functions **shadow** and **shadowing-import** may be used to declare shadowing symbols.

Name conflicts are detected when they become possible, that is, when the package structure is altered. There is no need to check for name conflicts during every name lookup.

The functions **use-package**, **import**, and **export** check for name conflicts. **use-package** makes the external symbols of the package being used accessible to the using package; each of these symbols is checked for name conflicts with the symbols already accessible. **import** adds a single symbol to the internals of a package, checking for a name conflict with an exist-

ing symbol either present in the package or accessible to it. **import** signals a name conflict error even if the conflict is with a shadowing symbol, the rationale being that the user has given two explicit and inconsistent directives. **export** makes a single symbol accessible to all the packages that use the package from which the symbol is exported. All of these packages are checked for name conflicts: **(export** *s p***)** does **(find-symbol (symbol-name** *s***)** *q***)** for each package *q* in **(package-used-by-list** *p***)**. Note that in the usual case of an **export** during the initial definition of a package, the result of **package-used-by-list** will be **nil** and the name-conflict checking will take negligible time.

The function **intern**, which is the one used most frequently by the Lisp reader for looking up names of symbols, does not need to do any name-conflict checking, because it never creates a new symbol if there is already an accessible symbol with the name given.

**shadow** and **shadowing-import** never signal a name-conflict error because the user, by calling these functions, has specified how any possible conflict is to be resolved. **shadow** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the specified name is accessible and, if so, whether it is directly present in the package or inherited. In the latter case, a new symbol is created to shadow it. **shadowing-import** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the same name is accessible; if so, it is shadowed by the new symbol, which implies that it must be uninterned if it was directly present in the package.

**unuse-package**, **unexport**, and **unintern** (when the symbol being uninterned is not a shadowing symbol) do not need to do any name-conflict checking because they only remove symbols from a package; they do not make any new symbols accessible.

Giving a shadowing symbol to **unintern** can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol **x**, and B and C each contain external symbols named **x**, then removing the shadowing symbol **x** from A will reveal a name conflict between **b:x** and **c:x** if those two symbols are distinct. In this case **unintern** will signal an error.

Aborting from a name-conflict error leaves the original symbol accessible. Package functions always signal name-conflict errors before making any change to the package structure. When multiple changes are to be made, however, for example when **export** is given a list of symbols, it is permissible

for the implementation to process each change separately, so that aborting from a name conflict caused by the second symbol in the list will not unexport the first symbol in the list. However, aborting from a name-conflict error caused by **export** of a single symbol will not leave that symbol accessible to some packages and inaccessible to others; with respect to each symbol processed, **export** behaves as if it were an atomic operation.

Continuing from a name-conflict error should offer the user a chance to resolve the name conflict in favor of either of the candidates. The package structure should be altered to reflect the resolution of the name conflict, via **shadowing-import**, **unintern**, or **unexport**.

A name conflict in **use-package** between a symbol directly present in the using package and an external symbol of the used package may be resolved in favor of the first symbol by making it a shadowing symbol, or in favor of the second symbol by uninterning the first symbol from the using package. The latter resolution is dangerous if the symbol to be uninterned is an external symbol of the using package, since it will cease to be an external symbol.

A name conflict in **use-package** between two external symbols inherited by the using package from other packages may be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol.

A name conflict in **export** between the symbol being exported and a symbol already present in a package that would inherit the newly exported symbol may be resolved in favor of the exported symbol by uninterning the other one, or in favor of the already-present symbol by making it a shadowing symbol.

A name conflict in **export** or **unintern** due to a package inheriting two distinct symbols with the same name from two other packages may be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol, just as with **use-package**.

A name conflict in **import** between the symbol being imported and a symbol inherited from some other package may be resolved in favor of the symbol being imported by making it a shadowing symbol, or in favor of the symbol already accessible by not doing the **import**. A name conflict in **import** with a symbol already present in the package may be resolved by uninterning that symbol, or by not doing the **import**.

Good user-interface style dictates that **use-package** and **export**, which can cause many name conflicts simultaneously, first check for all of the name conflicts before presenting any of them to the user. The user may then choose

to resolve all of them wholesale or to resolve each of them individually, the latter requiring a lot of interaction but permitting different conflicts to be resolved different ways.

Implementations may offer other ways of resolving name conflicts. For instance, if the symbols that conflict are not being used as objects but only as names for functions, it may be possible to "merge" the two symbols by putting the function definition onto both symbols. References to either symbol for purposes of calling a function would be equivalent. A similar merging operation can be done for variable values and for things stored on the property list. In Lisp Machine Lisp, for example, one can also *forward* the value, function, and property cells so that future changes to either symbol will propagate to the other one. Some other implementations are able to do this with value cells but not with property lists. Only the user can know whether this way of resolving a name conflict is adequate, because it will work only if the use of two non-**eq** symbols with the same name will not prevent the correct operation of the program. The value of offering symbol merging as a way of resolving name conflicts is that it can avoid the need to throw away the whole Lisp world, correct the package-definition forms that caused the error, and start over from scratch.

## 11.6 Built-in Packages

**common-lisp** The package named **common-lisp** contains the primitives of the ANSI Common Lisp system (as opposed to a Common Lisp system based on the 1984 specification). Its external symbols include all of the user-visible functions and global variables that are present in the ANSI Common Lisp system, such as **car**, **cdr**, and ***package***. Note, however, that the home package of such symbols is not necessarily the **common-lisp** package (this makes it easier for symbols such as **t** and **lambda** to be shared between the **common-lisp** package and another package, possibly one named **lisp**). Almost all other packages ought to use **common-lisp** so that these symbols will be accessible without qualification. This package has the nickname **cl**.

**common-lisp-user** The **common-lisp-user** package is, by default, the current package at the time an ANSI Common Lisp system starts up. This package uses the **common-lisp** package and has the nickname

**cl-user**. It may contain other implementation-dependent symbols and may use other implementation-specific packages.

**keyword** This package contains all of the keywords used by built-in or user-defined Lisp functions. Printed symbol representations that start with a colon are interpreted as referring to symbols in this package, which are always external symbols. All symbols in this package are treated as constants that evaluate to themselves, so that the user can type **:foo** instead of **':foo**.

X3J13 voted in January 1989 to modify the requirements on the built-in packages so as to limit what may appear in the **common-lisp** package and to lift the requirement that every implementation have a package named **system**. The details are as follows.

Not only must the **common-lisp** package in any given implementation contain all the external symbols prescribed by the standard; the **common-lisp** package moreover may not contain any external symbol that is not prescribed by the standard. However, the **common-lisp** package may contain additional internal symbols, depending on the implementation.

An external symbol of the **common-lisp** package may not have a function, macro, or special operator definition, or a top-level value, or a **special** proclamation, or a type definition, unless specifically permitted by the standard. Programmers may validly rely on this fact; for example, **fboundp** is guaranteed to be false for all external symbols of the **common-lisp** package except those explicitly specified in the standard to name functions, macros, and special operators. Similarly, **boundp** will be false of all such external symbols except those documented to be variables or constants.

Portable programs may use external symbols in the **common-lisp** package that are not documented to be constants or variables as names of local lexical variables with the presumption that the implementation has not proclaimed such variables to be special; this legitimizes the common practice of using such names as **list** and **string** as names for local variables.

A valid implementation may initially have properties on any symbol, or dynamically put new properties on symbols (even user-created symbols), as long as no property indicator used for this purpose is an external symbol of any package defined by the standard or a symbol that is accessible from the **common-lisp-user** package or any package defined by the user.

This vote eliminates the requirement that every implementation have a predefined package named **system**. An implementation may provide any

number of predefined packages; these should be described in the documentation for that implementation.

The **common-lisp-user** package may contain symbols not described by the standard and may use other implementation-specific packages.

X3J13 voted in March 1989 to restrict user programs from performing certain actions that might interfere with built-in facilities or interact badly with them. Except where explicitly allowed, the consequences are undefined if any of the following actions are performed on a symbol in the **common-lisp** package.

- binding or altering its value (lexically or dynamically)
- defining or binding it as a function
- defining or binding it as a macro
- defining it as a type specifier (**defstruct**, **defclass**, **deftype**)
- defining it as a structure (**defstruct**)
- defining it as a declaration
- design it as a symbol macro FIXME
- altering its print name
- altering its package
- tracing it
- declaring or proclaiming it special or lexical
- declaring or proclaiming its **type** or **ftype**
- removing it from the package **common-lisp**

X3J13 also voted in June 1989 to add to this list the item

- defining it as a compiler macro

If such a symbol is not globally defined as a variable or a constant, a user program is allowed to lexically bind it and declare the **type** of that binding.

If such a symbol is not defined as a function, macro, or special operator, a user program is allowed to (lexically) bind it as a function and to declare the **ftype** of that binding and to trace that binding.

If such a symbol is not defined as a function, macro, or special operator, a user program is allowed to (lexically) bind it as a macro.

As an example, the behavior of the code fragment

```
(flet ((open (filename &key direction)
         (format t "~%OPEN was called.")
         (open filename :direction direction)))
  (with-open-file (x "frob" :direction ':output)
    (format t "~%Was OPEN called?")))
```

is undefined. Even in a "reasonable" implementation, for example, the macro expansion of **with-open-file** might refer to the **open** function and might not. However, the preceding rules eliminate the burden of deciding whether an implementation is reasonable. The code fragment violates the rules; officially its behavior is therefore completely undefined, and that's that.

Note that "altering the property list" is not in the list of proscribed actions, so a user program *is* permitted to add properties to or remove properties from symbols in the **common-lisp** package.

## 11.7 Package System Functions and Variables

---

**Implementation note:** In the past, some Lisp compilers have read the entire file into Lisp before processing any of the forms. Other compilers have arranged for the loader to do all of its intern operations before evaluating any of the top-level forms. Neither of these techniques will work in a straightforward way in Common Lisp because of the presence of multiple packages.

---

For the functions described here, all optional arguments named *package* default to the current value of ***package***. Where a function takes an argument that is either a symbol or a list of symbols, an argument of **nil** is treated as an empty list of symbols. Any argument described as a package name may

be either a string or a symbol. If a symbol is supplied, its print name will be used as the package name; if a string is supplied, the user must take care to specify the same capitalization used in the package name, normally all uppercase.

*[Variable]* ***package***

The value of this variable must be a package; this package is said to be the current package. The initial value of ***package*** is the **user** package.

The functions **load** and **compile-file** rebind ***package*** to its current value. If some form in the file changes the value of ***package*** during loading or compilation, the old value will be restored when the loading is completed.

*[Function]* **make-package** *package-name* `&key` *:nicknames :use*

This creates and returns a new package with the specified package name. As described above, this argument may be either a string or a symbol. The **:nicknames** argument must be a list of strings to be used as alternative names for the package. Once again, the user may supply symbols in place of the strings, in which case the print names of the symbols are used. These names and nicknames must not conflict with any existing package names; if they do, a correctable error is signaled.

The **:use** argument is a list of packages or the names (strings or symbols) of packages whose external symbols are to be inherited by the new package. These packages must already exist. If not supplied, **:use** defaults to a list of one package, the **lisp** package.

*[Macro]* **in-package** name

This macro causes ***package*** to be set to the package named *name*, which must be a symbol or string. The *name* is not evaluated. An error is signaled if the package does not already exist. Everything this macro does is also performed at compile time if the call appears at top level.

**in-package** returns the new package, that is, the value of ***package*** after the operation has been executed.

*[Function]* **find-package** *name*

The *name* must be a string that is the name or nickname for a package. This argument may also be a symbol, in which case the symbol's print name

is used. The package with that name or nickname is returned; if no such package exists, **find-package** returns **nil**. The matching of names observes case (as in **string=**).

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **package-name** *package*

The argument must be a package. This function returns the string that names that package.

*package* argument may be either a package object or a package name (see section 11.2).

**package-name** returns **nil** instead of the package if the package has been removed. See **delete-package**.

*[Function]* **package-nicknames** *package*

The argument must be a package. This function returns the list of nickname strings for that package, not including the primary name.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **rename-package** *package new-name* `&optional` *new-nicknames*

The old name and all of the old nicknames of *package* are eliminated and are replaced by *new-name* and *new-nicknames*. The *new-name* argument is a string or symbol; the *new-nicknames* argument, which defaults to **nil**, is a list of strings or symbols.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **package-use-list** *package*

A list of other packages used by the argument package is returned.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **package-used-by-list** *package*

A list of other packages that use the argument package is returned.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **package-shadowing-symbols** *package*

A list is returned of symbols that have been declared as shadowing symbols in this package by **shadow** or **shadowing-import**. All symbols on this list are present in the specified package.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **list-all-packages**

This function returns a list of all packages that currently exist in the Lisp system.

*[Function]* **delete-package** *package*

The **delete-package** function deletes the specified *package* from all package system data structures. The *package* argument may be either a package or the name of a package.

If *package* is a name but there is currently no package of that name, a correctable error is signaled. Continuing from the error makes no deletion attempt but merely returns **nil** from the call to **delete-package**.

If *package* is a package object that has already been deleted, no error is signaled and no deletion is attempted; instead, **delete-package** immediately returns **nil**.

If the package specified for deletion is currently used by other packages, a correctable error is signaled. Continuing from this error, the effect of the function **unuse-package** is performed on all such other packages so as to remove their dependency on the specified package, after which **delete-package** proceeds to delete the specified package as if no other package had been using it.

If any symbol had the specified package as its home package before the call to **delete-package**, then its home package is unspecified (that is, the

contents of its package cell are unspecified) after the **delete-package** operation has been completed. Symbols in the deleted package are not modified in any other way.

The name and nicknames of the *package* cease to be recognized package names. The package object is still a package, but anonymous; **packagep** will be true of it, but **package-name** applied to it will return **nil**.

The effect of any other package operation on a deleted package object is undefined. In particular, an attempt to locate a symbol within a deleted package (using **intern** or **find-symbol**, for example) will have unspecified results.

**delete-package** returns **t** if the deletion succeeds, and **nil** otherwise.

*[Function]* **intern** *string* `&optional` *package*

The *package*, which defaults to the current package, is searched for a symbol with the name specified by the *string* argument. This search will include inherited symbols, as described in section 11.4. If a symbol with the specified name is found, it is returned. If no such symbol is found, one is created and is installed in the specified package as an internal symbol (as an external symbol if the package is the **keyword** package); the specified package becomes the home package of the created symbol.

X3J13 voted in March 1989 to specify that **intern** may in effect perform the search using a copy of the argument string in which some or all of the implementation-defined attributes have been removed from the characters of the string. It is implementation-dependent which attributes are removed.

Two values are returned. The first is the symbol that was found or created. The second value is **nil** if no pre-existing symbol was found, and takes on one of three values if a symbol was found:

**:internal** The symbol was directly present in the package as an internal symbol.

**:external** The symbol was directly present as an external symbol.

**:inherited** The symbol was inherited via **use-package** (which implies that the symbol is internal).

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **find-symbol** *string* `&optional` *package*

This is identical to **intern**, but it never creates a new symbol. If a symbol with the specified name is found in the specified package, directly or by inheritance, the symbol found is returned as the first value and the second value is as specified for **intern**. If the symbol is not accessible in the specified package, both values are **nil**.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **unintern** *symbol* `&optional` *package*

If the specified symbol is present in the specified *package*, it is removed from that package and also from the package's shadowing-symbols list if it is present there. Moreover, if the *package* is the home package for the symbol, the symbol is made to have no home package. Note that in some circumstances the symbol may continue to be accessible in the specified package by inheritance. **unintern** returns **t** if it actually removed a symbol, and **nil** otherwise.

**unintern** should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **export** *symbols* `&optional` *package*

The *symbols* argument should be a list of symbols, or possibly a single symbol. These symbols become accessible as external symbols in *package* (see section 11.4). **export** returns **t**.

By convention, a call to **export** listing all exported symbols is placed near the start of a file to advertise which of the symbols mentioned in the file are intended to be used by other programs.

*package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **unexport** *symbols* `&optional` *package*

The *symbols* argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in *package*. It is an error to

unexport a symbol from the **keyword** package (see section 11.4). **unexport** returns **t**.

The *package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **import** *symbols* `&optional` *package*

The argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in *package* and can therefore be referred to without having to use qualified-name (colon) syntax. **import** signals a correctable error if any of the imported symbols has the same name as some distinct symbol already accessible in the package (see section 11.4). **import** returns **t**.

If any symbol to be imported has no home package then **import** sets the home package of the symbol to the *package* to which the symbol is being imported.

The *package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **shadowing-import** *symbols* `&optional` *package*

This is like **import**, but it does not signal an error even if the importation of a symbol would shadow some symbol already accessible in the package. In addition to being imported, the symbol is placed on the shadowing-symbols list of *package* (see section 11.5). **shadowing-import** returns **t**.

**shadowing-import** should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

The *package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **shadow** *symbols* `&optional` *package*

The argument should be a list of symbols, or possibly a single symbol. The print name of each symbol is extracted, and the specified *package* is searched for a symbol of that name. If such a symbol is present in this package (directly, not by inheritance), then nothing is done. Otherwise, a new symbol is created with this print name, and it is inserted in the *package*

as an internal symbol. The symbol is also placed on the shadowing-symbols list of the *package* (see section 11.5). **shadow** returns **t**.

**shadow** should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

The *package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **use-package** *packages-to-use* `&optional` *package*

The *packages-to-use* argument should be a list of packages or package names, or possibly a single package or package name. These packages are added to the use-list of *package* if they are not there already. All external symbols in the packages to use become accessible in *package* as internal symbols (see section 11.4). It is an error to try to use the **keyword** package. **use-package** returns **t**.

The *package* argument may be either a package object or a package name (see section 11.2).

*[Function]* **unuse-package** *packages-to-unuse* `&optional` *package*

The *packages-to-unuse* argument should be a list of packages or package names, or possibly a single package or package name. These packages are removed from the use-list of *package*. **unuse-package** returns **t**.

The *package* argument may be either a package object or a package name (see section 11.2).

*[Macro]* **defpackage** defined-package-name {option}*

This creates a new package, or modifies an existing one, whose name is *defined-package-name*. The *defined-package-name* may be a string or a symbol; if it is a symbol, only its print name matters, and not what package, if any, the symbol happens to be in. The newly created or modified package is returned as the value of the **defpackage** form.

Each standard *option* is a list of a keyword (the name of the option) and associated arguments. No part of a **defpackage** form is evaluated. Except for the **:size** option, more than one option of the same kind may occur within the same **defpackage** form.

The standard options for **defpackage** are as follows. In every case, any option argument called *package-name* or *symbol-name* may be a string or a symbol; if it is a symbol, only its print name matters, and not what package, if any, the symbol happens to be in.

**(:size *integer*)** This specifies approximately the number of symbols expected to be in the package. This is purely an efficiency hint to the storage allocator, so that implementations using hash tables as part of the package data structure (the usual technique) will not have to incrementally expand the package as new symbols are added to it (for example, as a large file is read while "in" that package).

**(:nicknames {}* package-name)** The specified names become nicknames of the package being defined. If any of the specified nicknames already refers to an existing package, a continuable error is signaled exactly as for the function **make-package**.

**(:shadow {}* symbol-name)** Symbols with the specified names are created as shadows in the package being defined, just as with the function **shadow**.

**(:shadowing-import-from *package-name* {}* symbol-name)** Symbols with the specified names are located in the specified package. These symbols are imported into the package being defined, shadowing other symbols if necessary, just as with the function **shadowing-import**. In no case will symbols be created in a package other than the one being defined; a continuable error is signaled if for any *symbol-name* there is no symbol of that name accessible in the package named *package-name*.

**(:use {}* package-name)** The package being defined is made to "use" (inherit from) the packages specified by this option, just as with the function **use-package**. If no **:use** option is supplied, then option is unspecified.

**(:import-from *package-name* {}* symbol-name)** Symbols with the specified names are located in the specified package. These symbols are imported into the package being defined, just as with the function **import**. In no case will symbols be created in a package other than the one being defined; a continuable error is signaled if for any *symbol-name* there is no symbol of that name accessible in the package named *package-name*.

**(:intern {}* symbol-name)** Symbols with the specified names are located or created in the package being defined, just as with the function **in-**

**tern**. Note that the action of this option may be affected by a **:use** option, because an inherited symbol will be used in preference to creating a new one.

**(:export {}* symbol-name)** Symbols with the specified names are located or created in the package being defined and then exported, just as with the function **export**. Note that the action of this option may be affected by a **:use**, **:import-from**, or **:shadowing-import-from** option, because an inherited or imported symbol will be used in preference to creating a new one.

The order in which options appear in a **defpackage** form does not matter; part of the convenience of **defpackage** is that it sorts out the options into the correct order for processing. Options are processed in the following order:

1. **:shadow** and **:shadowing-import-from**
2. **:use**
3. **:import-from** and **:intern**
4. **:export**

Shadows are established first in order to avoid spurious name conflicts when use links are established. Use links must occur before importing and interning so that those operations may refer to normally inherited symbols rather than creating new ones. Exports are performed last so that symbols created by any of the other options, in particular, shadows and imported symbols, may be exported. Note that exporting an inherited symbol implicitly imports it first (see section 11.4).

If no package named *defined-package-name* already exists, **defpackage** creates it. If such a package does already exist, then no new package is created. The existing package is modified, if possible, to reflect the new definition. The results are undefined if the new definition is not consistent with the current state of the package.

An error is signaled if more than one **:size** option appears. Если опция **:size** указана более одного раза сигнализируется ошибка.

An error is signaled if the same **symbol-name** argument (in the sense of comparing names with **string=**) appears more than once among the arguments to all the **:shadow**, **:shadowing-import-from**, **:import-from**, and **:intern** options.

An error is signaled if the same **symbol-name** argument (in the sense of comparing names with **string=**) appears more than once among the arguments to all the **:intern** and **:export** options.

Other kinds of name conflicts are handled in the same manner that the underlying operations **use-package**, **import**, and **export** would handle them.

Implementations may support other **defpackage** options. Every implementation should signal an error on encountering a **defpackage** option it does not support.

The function **compile-file** should treat top-level **defpackage** forms in the same way it would treat top-level calls to package-affecting functions (as described at the beginning of section 11.7).

Here is an example of a call to **defpackage** that "plays it safe" by using only strings as names.

```
(cl:defpackage "MY-VERY-OWN-PACKAGE"
  (:size 496)
  (:nicknames "MY-PKG" "MYPKG" "MVOP")
  (:use "COMMON-LISP")
  (:shadow "CAR" "CDR")
  (:shadowing-import-from "BRAND-X-LISP" "CONS")
  (:import-from "BRAND-X-LISP" "GC" "BLINK-FRONT-PANEL-LIGHTS")
  (:export "EQ" "CONS" "MY-VERY-OWN-FUNCTION"))
```

The preceding **defpackage** example is designed to operate correctly even if the package current when the form is read happens not to "use" the **common-lisp** package. (Note the use in this example of the nickname **cl** for the **common-lisp** package.) Moreover, neither reading in nor evaluating this **defpackage** form will ever create any symbols in the current package. Note too the use of uppercase letters in the strings.

Here, for the sake of contrast, is a rather similar use of **defpackage** that "plays the whale" by using all sorts of permissible syntax.

```
(defpackage my-very-own-package
  (:export :EQ common-lisp:cons my-very-own-function)
  (:nicknames "MY-PKG" #:MyPkg)
  (:use "COMMON-LISP")
  (:shadow "CAR")
  (:size 496)
  (:nicknames mvop)
```

```
(:import-from "BRAND-X-LISP" "GC" Blink-Front-Panel-Lights)
(:shadow common-lisp::cdr)
(:shadowing-import-from "BRAND-X-LISP" CONS))
```

This example has exactly the same effect on the newly created package but may create useless symbols in other packages. The use of explicit package tags is particularly confusing; for example, this **defpackage** form will cause the symbol **cdr** to be shadowed *in the new package*; it will not be shadowed in the package **common-lisp**. The fact that the name "**CDR**" was specified by a package-qualified reference to a symbol in the **common-lisp** package is a red herring. The moral is that the syntactic flexibility of **defpackage**, as in other parts of Common Lisp, yields considerable convenience when used with commonsense competence, but unutterable confusion when used with Malthusian profusion.

---

**Implementation note:** An implementation of **defpackage** might choose to transform all the *package-name* and *symbol-name* arguments into strings at macro expansion time, rather than at the time the resulting expansion is executed, so that even if source code is expressed in terms of strange symbols in the **defpackage** form, the binary file resulting from compiling the source code would contain only strings. The purpose of this is simply to minimize the creation of useless symbols in production code. This technique is permitted as an implementation strategy but is not a behavior required by the specification of **defpackage**.

---

Note that **defpackage** is not capable by itself of defining mutually recursive packages, for example two packages each of which uses the other. However, nothing prevents one from using **defpackage** to perform much of the initial setup and then using functions such as **use-package**, **import**, and **export** to complete the links.

The purpose of **defpackage** is to encourage the user to put the entire definition of a package and its relationships to other packages in a single place. It may also encourage the designer of a large system to place the definitions of all relevant packages into a single file (say) that can be loaded before loading or compiling any code that depends on those packages. Such a file, if carefully constructed, can simply be loaded into the **common-lisp-user** package.

Implementations and programming environments may also be better able to support the programming process (if only by providing better error checking) through global knowledge of the intended package setup.

*[Function]* **find-all-symbols** *string-or-symbol*

**find-all-symbols** searches every package in the Lisp system to find every symbol whose print name is the specified string. A list of all such symbols found is returned. This search is case-sensitive. If the argument is a symbol, its print name supplies the string to be searched for.

*[Macro]* **do-symbols** (var [package [result-form]])
{declaration}* {tag | statement}*

**do-symbols** provides straightforward iteration over the symbols of a package. The body is performed once for each symbol accessible in the *package*, in no particular order, with the variable *var* bound to the symbol. Then *result-form* (a single form, *not* an implicit **progn**) is evaluated, and the result is the value of the **do-symbols** form. (When the *result-form* is evaluated, the control variable *var* is still bound and has the value **nil**.) If the *result-form* is omitted, the result is **nil**. **return** may be used to terminate the iteration prematurely. If execution of the body affects which symbols are contained in the *package*, other than possibly to remove the symbol currently the value of *var* by using **unintern**, the effects are unpredictable.

The *package* argument may be either a package object or a package name (see section 11.2).

X3J13 voted in March 1988 to specify that the body of a **do-symbols** form may be executed more than once for the same accessible symbol, and users should take care to allow for this possibility.

The point is that the same symbol might be accessible via more than one chain of inheritance, and it is implementationally costly to eliminate such duplicates. Here is an example:

```
(setq *a* (make-package 'a))      ;Implicitly uses package common-lisp
(setq *b* (make-package 'b))      ;Implicitly uses package common-lisp
(setq *c* (make-package 'c :use '(a b)))

(do-symbols (x *c*) (print x))    ;Symbols in package common-lisp
                                  ; might be printed once or twice here
```

X3J13 voted in January 1989 to restrict user side effects; see section 7.9. Note that the **loop** construct provides a kind of **for** clause that can iterate

over the symbols of a package (see chapter 26).

*[Macro]* **do-external-symbols** (var [package [result]])
{declaration}* {tag | statement}*

**do-external-symbols** is just like **do-symbols**, except that only the external symbols of the specified package are scanned.

The clarification voted by X3J13 in March 1988 for **do-symbols** , regarding redundant executions of the body for the same symbol, applies also to **do-external-symbols**.

The *package* argument may be either a package object or a package name (see section 11.2).

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

*[Macro]* **do-all-symbols** (var [result-form])
{declaration}* {tag | statement}*

This is similar to **do-symbols** but executes the body once for every symbol contained in every package. (This will not process every symbol whatsoever, because a symbol not accessible in any package will not be processed. Normally, uninterned symbols are not accessible in any package.) It is *not* in general the case that each symbol is processed only once, because a symbol may appear in many packages.

The clarification voted by X3J13 in March 1988 for **do-symbols** , regarding redundant executions of the body for the same symbol, applies also to **do-all-symbols**.

The *package* argument may be either a package object or a package name (see section 11.2).

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

*[Macro]* **with-package-iterator** (mname package-list {symbol-type}+)
{form}*

The name *mname* is bound and defined as if by **macrolet**, with the body *form*s as its lexical scope, to be a "generator macro" such that each invocation of **(*mname*)** will return a symbol and that successive invocations will eventually deliver, one by one, all the symbols from the packages that are elements of the list that is the value of the expression *package-list* (which is evaluated exactly once).

Each element of the *package-list* value may be either a package or the name of a package. As a further convenience, if the *package-list* value is itself a package or the name of a package, it is treated as if a singleton list

containing that value had been provided. If the *package-list* value is **nil**, it is considered to be an empty list of packages.

At each invocation of the generator macro, there are two possibilities. If there is yet another unprocessed symbol, then four values are returned: **t**, the symbol, a keyword indicating the accessibility of the symbol within the package (see below), and the package from which the symbol was accessed. If there are no more unprocessed symbols in the list of packages, then one value is returned: **nil**.

When the generator macro returns a symbol as its second value, the fourth value is always one of the packages present or named in the *package-list* value, and the third value is a keyword indicating accessibility: **:internal** means present in the package and not exported; **:external** means present and exported; and **:inherited** means not present (thus not shadowed) but inherited from some package used by the package that is the fourth value.

Each *symbol-type* in an invocation of **with-package-iterator** is not evaluated. More than one may be present; their order does not matter. They indicate the accessibility types of interest. A symbol is not returned by the generator macro unless its actual accessibility matches one of the *symbol-type* indicators. The standard *symbol-type* indicators are **:internal**, **:external**, and **:inherited**, but implementations are permitted to extend the syntax of **with-package-iterator** by recognizing additional symbol accessibility types. An error is signaled if no *symbol-type* is supplied, or if any supplied *symbol-type* is not recognized by the implementation.

The order in which symbols are produced by successive invocations of the generator macro is not necessarily correlated in any way with the order of the packages in the *package-list*. When more than one package is in the *package-list*, symbols accessible from more than one package may be produced once or more than once. Even when only one package is specified, symbols inherited in multiple ways via used packages may be produced once or more than once.

The implicit interior state of the iteration over the list of packages and the symbols within them has dynamic extent. It is an error to invoke the generator macro once the **with-package-iterator** form has been exited.

Any number of invocations of **with-package-iterator** and related macros may be nested, and the generator macro of an outer invocation may be called from within an inner invocation (provided, of course, that its name is visible or otherwise made available).

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

---

**Rationale:** This facility is a bit more flexible in some ways than **do-symbols** and friends. In particular, it makes it possible to implement **loop** clauses for iterating over packages in a way that is both portable and efficient (see chapter 26).

---