11. Packages

11.1 Package Concepts

11.1.1 Introduction to Packages

A *package* establishes a mapping from names to *symbols*. At any given time, one *package* is current. The *current package* is the one that is the *value* of ***package***. When using the *Lisp reader*, it is possible to refer to *symbols* in *packages* other than the current one through the use of *package prefixes* in the printed representation of the *symbol*.

The next figure lists some *defined names* that are applicable to *packages*. Where an *operator* 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 *package* argument may be either a *string*, a *symbol*, or a *package*. If a *symbol* is supplied, its name will be used as the *package* name.

```
*modules*            import                    provide
*package*            in-package                rename-package
defpackage           intern                    require
do-all-symbols       list-all-packages         shadow
do-external-symbols  make-package              shadowing-import
do-symbols           package-name              unexport
export               package-nicknames         unintern
find-all-symbols     package-shadowing-symbols unuse-package
find-package         package-use-list          use-package
find-symbol          package-used-by-list
```

Figure 11-1. Some Defined Names related to Packages

11.1.1.1 Package Names and Nicknames

Each *package* has a *name* (a *string*) and perhaps some *nicknames* (also *strings*). These are assigned when the *package* is created and can be changed later.

There is a single namespace 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*. **rename-package** removes a *package*'s current *name* and *nicknames* and replaces them with new ones specified by the caller.

11.1.1.2 Symbols in a Package

11.1.1.2.1 Internal and External Symbols

The mappings in a *package* are divided into two classes, external and internal. The *symbols* targeted by these different mappings are called *external symbols* and *internal symbols* of the *package*. Within a *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*. *Symbols* become *external symbols* of a given *package* if they have been *exported* from that *package*.

A *symbol* has the same *name* no matter what *package* it is *present* in, but it might be an *external symbol* of some *packages* and an *internal symbol* of others.

11.1.1.2.2 Package Inheritance

Packages can be built up in layers. From one point of view, a *package* is a single collection of mappings from *strings* into *internal symbols* and *external symbols*. However, some of these mappings might be established within the *package* itself, while other mappings are inherited from other *packages* via **use-package**. A *symbol* is said to be *present* in a *package* if the mapping is in the *package* itself and is not inherited from somewhere else.

There is no way to inherit the *internal symbols* of another *package*; to refer to an *internal symbol* using the *Lisp reader*, a *package* containing the *symbol* must be made to be the *current package*, a *package prefix* must be used, or the *symbol* must be *imported* into the *current package*.

11.1.1.2.3 Accessibility of Symbols in a Package

A *symbol* becomes *accessible* in a *package* if that is its *home package* when it is created, or if it is *imported* into that *package*, or by inheritance via **use-package**.

If a *symbol* is *accessible* in a *package*, it can be referred to when using the *Lisp reader* without a *package prefix* when that *package* is the *current package*, regardless of whether it is *present* or inherited.

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

-- Any individual *symbol* can be added to a *package* by use of **import**. After the call to **import** the *symbol* is *present* in the importing *package*. The status of the *symbol* in the *package* it came from (if any) is unchanged, and the *home package* for this *symbol* is unchanged. Once *imported*, a *symbol* is *present* in the importing *package* and can be removed only by calling **unintern**.

A *symbol* is *shadowed*[3] by another *symbol* in some *package* if the first *symbol* would be *accessible* by inheritance if not for the presence of the second *symbol*. See **shadowing-import**.

-- The second mechanism for making *symbols* from one *package accessible* in another is provided by **use-package**. All of the *external symbols* of the used *package* are inherited by the using *package*. The *function* **unuse-package** undoes the effects of a previous **use-package**.

11.1.1.2.4 Locating a Symbol in a Package

When a *symbol* is to be located in a given *package* the following occurs:

-- The *external symbols* and *internal symbols* of the *package* are searched for the *symbol*.
-- The *external symbols* of the used *packages* are searched in some unspecified order. The order does not matter; see the rules for handling name conflicts listed below.

11.1.1.2.5 Prevention of Name Conflicts in Packages

Within one *package*, any particular name can refer to at most one *symbol*. A name conflict is said to occur when there would be more than one candidate *symbol*. Any time a name conflict is about to occur, a *correctable error* is signaled.

The following rules apply to name conflicts:

-- Name conflicts are detected when they become possible, that is, when the package structure is altered. Name conflicts are not checked during every name lookup.
-- If the *same symbol* is *accessible* to a *package* through more than one path, there is no name conflict. A *symbol* cannot conflict with itself. Name conflicts occur only between *distinct symbols* with the same name (under **string=**).

-- 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* in 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 exception involving **import**). See **shadow** and **shadowing-import**.
-- The functions **use-package**, **import**, and **export** check for name conflicts.
-- **shadow** and **shadowing-import** never signal a name-conflict error.
-- **unuse-package** and **unexport** do not need to do any name-conflict checking. **unintern** does name-conflict checking only when a *symbol* being *uninterned* is a *shadowing symbol*.
-- Giving a shadowing symbol to **unintern** can uncover a name conflict that had previously been resolved by the shadowing.
-- Package functions signal name-conflict errors of *type* **package-error** before making any change to the package structure. When multiple changes are to be made, it is permissible for the implementation to process each change separately. For example, when **export** is given a *list* of *symbols*, aborting from a name conflict caused by the second *symbol* in the *list* might still export the first *symbol* in the *list*. However, a name-conflict error caused by **export** of a single *symbol* will be signaled before that *symbol*'s *accessibility* in any *package* is changed.
-- Continuing from a name-conflict error must 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 present* in the using *package* and an *external symbol* of the used *package* is 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*.
-- A name conflict in **export** or **unintern** due to a *package*'s inheriting two *distinct symbols* with the *same name* (under **string=**) from two other *packages* can 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**.

11.1.2 Standardized Packages

This section describes the *packages* that are available in every *conforming implementation*. A summary of the *names* and *nicknames* of those *standardized packages* is given in the next figure.

Name	Nicknames
COMMON-LISP	CL
COMMON-LISP-USER	CL-USER
KEYWORD	none

Figure 11-2. Standardized Package Names

11.1.2.1 The COMMON-LISP Package

The COMMON-LISP package contains the primitives of the Common Lisp system as defined by this specification. Its *external symbols* include all of the *defined names* (except for *defined names* in the KEYWORD package) that are present in the Common Lisp system, such as **car**, **cdr**, ***package***, etc. The COMMON-LISP package has the *nickname* CL.

The COMMON-LISP package has as *external symbols* those symbols enumerated in the figures in Section 1.9 (Symbols in the COMMON-LISP Package), and no others. These *external symbols* are *present* in the COMMON-LISP package but their *home package* need not be the COMMON-LISP package.

For example, the symbol HELP cannot be an *external symbol* of the COMMON-LISP package because it is not mentioned in Section 1.9 (Symbols in the COMMON-LISP Package). In contrast, the *symbol* **variable** must be an *external symbol* of the COMMON-LISP package even though it has no definition because it is listed in that section (to support its use as a valid second *argument* to the *function* **documentation**).

The COMMON-LISP package can have additional *internal symbols*.

11.1.2.1.1 Constraints on the COMMON-LISP Package for Conforming Implementations

In a *conforming implementation*, an *external symbol* of the COMMON-LISP package can have a *function*, *macro*, or *special operator* definition, a *global variable* definition (or other status as a *dynamic variable* due to a **special** *proclamation*), or a *type* definition only if explicitly permitted in this standard. For example, **fboundp** *yields false* for any *external symbol* of the COMMON-LISP package that is not the *name* of a *standardized function*, *macro* or *special operator*, and **boundp** returns *false* for any *external symbol* of the COMMON-LISP package that is not the *name* of a *standardized global variable*. It also follows that *conforming programs* can use *external symbols* of the COMMON-LISP package as the *names* of local *lexical variables* with confidence that those *names* have not been *proclaimed* **special** by the *implementation* unless those *symbols* are *names* of *standardized global variables*.

A *conforming implementation* must not place any *property* on an *external symbol* of the COMMON-LISP package using a *property indicator* that is either an *external symbol* of any *standardized package* or a *symbol* that is otherwise *accessible* in the COMMON-LISP-USER package.

11.1.2.1.2 Constraints on the COMMON-LISP Package for Conforming Programs

Except where explicitly allowed, the consequences are undefined if any of the following actions are performed on an *external symbol* of the COMMON-LISP package:

1. *Binding* or altering its value (lexically or dynamically). (Some exceptions are noted below.)
2. Defining, undefining, or *binding* it as a *function*. (Some exceptions are noted below.)
3. Defining, undefining, or *binding* it as a *macro* or *compiler macro*. (Some exceptions are noted below.)
4. Defining it as a *type specifier* (via **defstruct**, **defclass**, **deftype**, **define-condition**).
5. Defining it as a structure (via **defstruct**).
6. Defining it as a *declaration* with a **declaration** *proclamation*.
7. Defining it as a *symbol macro*.
8. Altering its *home package*.
9. Tracing it (via **trace**).
10. Declaring or proclaiming it **special** (via **declare**, **declaim**, or **proclaim**).
11. Declaring or proclaiming its **type** or **ftype** (via **declare**, **declaim**, or **proclaim**). (Some exceptions are noted below.)
12. Removing it from the COMMON-LISP package.
13. Defining a *setf expander* for it (via **defsetf** or **define-setf-method**).
14. Defining, undefining, or binding its *setf function name*.
15. Defining it as a *method combination* type (via **define-method-combination**).
16. Using it as the class-name argument to **setf** of **find-class**.
17. Binding it as a *catch tag*.
18. Binding it as a *restart name*.
19. Defining a *method* for a *standardized generic function* which is *applicable* when all of the *arguments* are *direct instances* of *standardized classes*.

11.1.2.1.2.1 Some Exceptions to Constraints on the COMMON-LISP Package for Conforming Programs

If an *external symbol* of the COMMON-LISP package is not globally defined as a *standardized dynamic variable* or *constant variable*, it is allowed to lexically *bind* it and to declare the **type** of that *binding*, and it is allowed to locally *establish* it as a *symbol macro* (e.g., with **symbol-macrolet**).

Unless explicitly specified otherwise, if an *external symbol* of the COMMON-LISP package is globally defined as a *standardized dynamic variable*, it is permitted to *bind* or *assign* that *dynamic variable* provided that the "Value Type" constraints on the *dynamic variable* are maintained, and that the new *value* of the *variable* is consistent with the stated purpose of the *variable*.

If an *external symbol* of the COMMON-LISP package is not defined as a *standardized function*, *macro*, or *special operator*, it is allowed to lexically *bind* it as a *function* (e.g., with **flet**), to declare the **ftype** of that *binding*, and (in *implementations* which provide the ability to do so) to **trace** that *binding*.

If an *external symbol* of the COMMON-LISP package is not defined as a *standardized function*, *macro*, or *special operator*, it is allowed to lexically *bind* it as a *macro* (e.g., with **macrolet**).

If an *external symbol* of the COMMON-LISP package is not defined as a *standardized function*, *macro*, or *special operator*, it is allowed to lexically *bind* its *setf function name* as a *function*, and to declare the **ftype** of that *binding*.

11.1.2.2 The COMMON-LISP-USER Package

The COMMON-LISP-USER package is the *current package* when a Common Lisp system starts up. This *package uses* the COMMON-LISP package. The COMMON-LISP-USER package has the *nickname* CL-USER. The COMMON-LISP-USER package can have additional *symbols interned* within it; it can *use* other *implementation-defined packages*.

11.1.2.3 The KEYWORD Package

The KEYWORD package contains *symbols*, called *keywords*[1], that are typically used as special markers in *programs* and their associated data *expressions*[1].

Symbol tokens that start with a *package marker* are parsed by the *Lisp reader* as *symbols* in the KEYWORD package; see Section 2.3.4 (Symbols as Tokens). This makes it notationally convenient to use *keywords* when communicating between programs in different *packages*. For example, the mechanism for passing *keyword parameters* in a *call* uses *keywords*[1] to name the corresponding *arguments*; see Section 3.4.1 (Ordinary Lambda Lists).

Symbols in the KEYWORD package are, by definition, of *type* **keyword**.

11.1.2.3.1 Interning a Symbol in the KEYWORD Package

The KEYWORD package is treated differently than other *packages* in that special actions are taken when a *symbol* is *interned* in it. In particular, when a *symbol* is *interned* in the KEYWORD package, it is automatically made to be an *external symbol* and is automatically made to be a *constant variable* with itself as a *value*.

11.1.2.3.2 Notes about The KEYWORD Package

It is generally best to confine the use of *keywords* to situations in which there are a finitely enumerable set of names to be selected between. For example, if there were two states of a light switch, they might be called :on and :off.

In situations where the set of names is not finitely enumerable (i.e., where name conflicts might arise) it is frequently best to use *symbols* in some *package* other than KEYWORD so that conflicts will be naturally avoided. For example, it is generally not wise for a *program* to use a *keyword*[1] as a *property indicator*, since if there were

ever another *program* that did the same thing, each would clobber the other's data.

11.1.2.4 Implementation-Defined Packages

Other, *implementation-defined packages* might be present in the initial Common Lisp environment.

It is recommended, but not required, that the documentation for a *conforming implementation* contain a full list of all *package* names initially present in that *implementation* but not specified in this specification. (See also the *function* **list-all-packages**.)