Programming Language—Common Lisp

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*.

Figure 11–1 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 Figure 11–2.

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 1.8 (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 1.8 (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 or *binding* it as a *function*. (Some exceptions are noted below.)

3. Defining 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 (via **declare**, **declaim**, or **proclaim**) **special**.

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*.

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**).

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**.)

package *System Class*

Class Precedence List:

package, **t**

Description:

A *package* is a *namespace* that maps *symbol names* to *symbols*; see Section 11.1 (Package Concepts).

See Also:

Section 11.1 (Package Concepts), Section 22.1.3.16 (Printing Other Objects), Section 2.3.4 (Symbols as Tokens)

export *Function*

Syntax:

export *symbols* `&optional` *package* → **t**

Arguments and Values:

symbols—a *designator* for a *list* of *symbols*.

package—a *package designator*. The default is the *current package*.

Description:

export makes one or more *symbols* that are *accessible* in *package* (whether directly or by inheritance) be *external symbols* of that *package*.

If any of the *symbols* is already *accessible* as an *external symbol* of *package*, **export** has no effect on that *symbol*. If the *symbol* is *present* in *package* as an internal symbol, it is simply changed to external status. If it is *accessible* as an *internal symbol* via **use-package**, it is first *imported* into *package*, then *exported*. (The *symbol* is then *present* in the *package* whether or not *package* continues to use the *package* through which the *symbol* was originally inherited.)

export makes each *symbol accessible* to all the *packages* that use *package*. 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** is **nil** and the name-conflict checking takes negligible time. When multiple changes are to be made, 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 any but the first *symbol* in the *list* does not unexport the first *symbol* in the *list*. However, aborting from a name-conflict error caused by **export** of one of *symbols* does not leave that *symbol accessible* to some *packages* and *inaccessible* to others; with respect to each of *symbols* processed, **export** behaves as if it were as an atomic operation.

A name conflict in **export** between one of *symbols* 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.

Examples:

```
(make-package 'temp :use nil) → #<PACKAGE "TEMP">
(use-package 'temp) → T
(intern "TEMP-SYM" 'temp) → TEMP::TEMP-SYM, NIL
(find-symbol "TEMP-SYM") → NIL, NIL
(export (find-symbol "TEMP-SYM" 'temp) 'temp) → T
(find-symbol "TEMP-SYM") → TEMP-SYM, :INHERITED
```

Side Effects:

The package system is modified.

Affected By:

Accessible symbols.

Exceptional Situations:

If any of the *symbols* is not *accessible* at all in *package*, an error of *type* **package-error** is signaled that is *correctable* by permitting the *user* to interactively specify whether that *symbol* should be *imported*.

See Also:

import, **unexport**, Section 11.1 (Package Concepts)

find-symbol *Function*

Syntax:

find-symbol *string* `&optional` *package* → *symbol, status*

Arguments and Values:

string—a *string*.

package—a *package designator*. The default is the *current package*.

symbol—a *symbol* accessible in the *package*, or **nil**.

status—one of `:inherited`, `:external`, `:internal`, or **nil**.

find-symbol

Description:

find-symbol locates a *symbol* whose *name* is *string* in a *package*. If a *symbol* named *string* is found in *package*, directly or by inheritance, the *symbol* found is returned as the first value; the second value is as follows:

`:internal`

If the *symbol* is *present* in *package* as an *internal symbol*.

`:external`

If the *symbol* is *present* in *package* as an *external symbol*.

`:inherited`

If the *symbol* is inherited by *package* through **use-package**, but is not *present* in *package*.

If no such *symbol* is *accessible* in *package*, both values are **nil**.

Examples:

```
 (find-symbol "NEVER-BEFORE-USED") → NIL, NIL
 (find-symbol "NEVER-BEFORE-USED") → NIL, NIL
 (intern "NEVER-BEFORE-USED") → NEVER-BEFORE-USED, NIL
 (intern "NEVER-BEFORE-USED") → NEVER-BEFORE-USED, :INTERNAL
 (find-symbol "NEVER-BEFORE-USED") → NEVER-BEFORE-USED, :INTERNAL
 (find-symbol "never-before-used") → NIL, NIL
 (find-symbol "CAR" 'common-lisp-user) → CAR, :INHERITED
 (find-symbol "CAR" 'common-lisp) → CAR, :EXTERNAL
 (find-symbol "NIL" 'common-lisp-user) → NIL, :INHERITED
 (find-symbol "NIL" 'common-lisp) → NIL, :EXTERNAL
 (find-symbol "NIL" (prog1 (make-package "JUST-TESTING" :use '())
                           (intern "NIL" "JUST-TESTING")))
→ JUST-TESTING::NIL, :INTERNAL
 (export 'just-testing::nil 'just-testing)
 (find-symbol "NIL" 'just-testing) → JUST-TESTING:NIL, :EXTERNAL
 (find-symbol "NIL" "KEYWORD")
→ NIL, NIL
or
→ :NIL, :EXTERNAL
 (find-symbol (symbol-name :nil) "KEYWORD") → :NIL, :EXTERNAL
```

Affected By:

intern, **import**, **export**, **use-package**, **unintern**, **unexport**, **unuse-package**

See Also:

intern, **find-all-symbols**

Notes:

find-symbol is operationally equivalent to **intern**, except that it never creates a new *symbol*.

find-package *Function*

Syntax:

find-package *name* → *package*

Arguments and Values:

name—a *package designator*.

package—a *package object* or **nil**.

Description:

find-package locates and returns the *package* whose name or nickname is *name*. If *name* is a *package object*, that *package object* is returned. If there is no such *package*, **find-package** returns **nil**.

The **find-package** search is case sensitive.

Examples:

```
(find-package 'common-lisp) → #<PACKAGE "COMMON-LISP">
(find-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(find-package 'not-there) → NIL
```

Affected By:

The set of *packages* created by the *implementation*.

defpackage, **delete-package**, **make-package**, **rename-package**

See Also:

make-package

find-all-symbols *Function*

Syntax:

find-all-symbols *string* → *symbols*

Arguments and Values:

string—a *symbol name designator*.

symbols—a *list* of *symbols*.

Description:

find-all-symbols searches every *registered package* for *symbols* that have a *name* that is the *same* (under **string=**) as *string*. A *list* of all such *symbols* is returned. Whether or how the *list* is ordered is *implementation-dependent*.

Examples:

```
 (find-all-symbols 'car)
→ (CAR)
or→ (CAR VEHICLES:CAR)
or→ (VEHICLES:CAR CAR)
 (intern "CAR" (make-package 'temp :use nil)) → TEMP::CAR, NIL
 (find-all-symbols 'car)
→ (TEMP::CAR CAR)
or→ (CAR TEMP::CAR)
or→ (TEMP::CAR CAR VEHICLES:CAR)
or→ (CAR TEMP::CAR VEHICLES:CAR)
```

See Also:

find-symbol

import *Function*

Syntax:

import *symbols* &optional *package* → t

Arguments and Values:

symbols—a *designator* for a *list* of *symbols*.

package—a *package designator*. The default is the *current package*.

Description:

import adds *symbol* or *symbols* to the internals of *package*, checking for name conflicts with existing *symbols* either *present* in *package* or *accessible* to it. Once the *symbols* have been *imported*, they may be referenced in the *importing package* without the use of a *package prefix* when using the *Lisp reader*.

A name conflict in **import** between the *symbol* being imported and a symbol inherited from some other *package* can 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**.

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. If any *symbol* to be *imported* has no home package (*i.e.*, `(symbol-package` *symbol*`)` → `nil`), **import** sets the *home package* of the *symbol* to *package*.

If the *symbol* is already *present* in the importing *package*, **import** has no effect.

Examples:

```
(import 'common-lisp::car (make-package 'temp :use nil)) → T
(find-symbol "CAR" 'temp) → CAR, :INTERNAL
(find-symbol "CDR" 'temp) → NIL, NIL
```

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 `buffer` is then *present* in the *current package*.

Side Effects:

The package system is modified.

Affected By:

Current state of the package system.

Exceptional Situations:

import signals a *correctable* error of *type* **package-error** if any of the *symbols* to be *imported* has the *same name* (under **string=**) as some distinct *symbol* (under **eql**) already *accessible* in the *package*, even if the conflict is with a *shadowing symbol* of the *package*.

See Also:

shadow, **export**

list-all-packages *Function*

Syntax:

list-all-packages ⟨*no arguments*⟩ → *packages*

Arguments and Values:

packages—a *list* of *package objects*.

Description:

list-all-packages returns a *fresh list* of all *registered packages*.

Examples:

```
(let ((before (list-all-packages)))
   (make-package 'temp)
   (set-difference (list-all-packages) before)) → (#<PACKAGE "TEMP">)
```

Affected By:

defpackage, **delete-package**, **make-package**

rename-package *Function*

Syntax:

rename-package *package new-name* &optional *new-nicknames* → *package-object*

Arguments and Values:

package—a *package designator*.

new-name—a *package designator*.

new-nicknames—a *list* of *package name designators*. The default is the *empty list*.

package-object—the renamed *package object*.

Description:

Replaces the name and nicknames of *package*. The old name and all of the old nicknames of *package* are eliminated and are replaced by *new-name* and *new-nicknames*.

The consequences are undefined if *new-name* or any *new-nickname* conflicts with any existing package names.

Examples:

```
(make-package 'temporary :nicknames '("TEMP")) → #<PACKAGE "TEMPORARY">
(rename-package 'temp 'ephemeral) → #<PACKAGE "EPHEMERAL">
(package-nicknames (find-package 'ephemeral)) → ()
(find-package 'temporary) → NIL
(rename-package 'ephemeral 'temporary '(temp fleeting))
→ #<PACKAGE "TEMPORARY">
(package-nicknames (find-package 'temp)) → ("TEMP" "FLEETING")
```

See Also:

make-package

shadow *Function*

Syntax:

shadow *symbol-names* &optional *package* → t

Arguments and Values:

symbol-names—a *designator* for a *list* of *symbol name designators*.

package—a *package designator*. The default is the *current package*.

Description:

shadow assures that *symbols* with names given by *symbol-names* are *present* in the *package*.

Specifically, *package* is searched for *symbols* with the *names* supplied by *symbol-names*. For each such *name*, if a corresponding *symbol* is not *present* in *package* (directly, not by inheritance), then a corresponding *symbol* is created with that *name*, and inserted into *package* as an *internal symbol*. The corresponding *symbol*, whether pre-existing or newly created, is then added, if not already present, to the *shadowing symbols list* of *package*.

Examples:

```
(package-shadowing-symbols (make-package 'temp)) → NIL
(find-symbol 'car 'temp) → CAR, :INHERITED
(shadow 'car 'temp) → T
(find-symbol 'car 'temp) → TEMP::CAR, :INTERNAL
(package-shadowing-symbols 'temp) → (TEMP::CAR)

(make-package 'test-1) → #<PACKAGE "TEST-1">
(intern "TEST" (find-package 'test-1)) → TEST-1::TEST, NIL
(shadow 'test-1::test (find-package 'test-1)) → T
```

```
(shadow 'TEST (find-package 'test-1)) → T
(assert (not (null (member 'test-1::test (package-shadowing-symbols
                                          (find-package 'test-1))))))

(make-package 'test-2) → #<PACKAGE "TEST-2">
(intern "TEST" (find-package 'test-2)) → TEST-2::TEST, NIL
(export 'test-2::test (find-package 'test-2)) → T
(use-package 'test-2 (find-package 'test-1))    ;should not error
```

Side Effects:

shadow changes the state of the package system in such a way that the package consistency rules do not hold across the change.

Affected By:

Current state of the package system.

See Also:

package-shadowing-symbols, Section 11.1 (Package Concepts)

Notes:

If a *symbol* with a name in *symbol-names* already exists in *package*, but by inheritance, the inherited symbol becomes *shadowed*$_3$ by a newly created *internal symbol*.

shadowing-import *Function*

Syntax:

shadowing-import *symbols* &optional *package* → t

Arguments and Values:

symbols—a *designator* for a *list* of *symbols*.

package —a *package designator*. The default is the *current package*.

Description:

shadowing-import is like **import**, but it does not signal an error even if the importation of a *symbol* would shadow some *symbol* already *accessible* in *package*.

shadowing-import inserts each of *symbols* into *package* as an internal symbol, regardless of whether another *symbol* of the same name is shadowed by this action. If a different *symbol* of the same name is already *present* in *package*, that *symbol* is first *uninterned* from *package*. The new *symbol* is added to *package*'s shadowing-symbols list.

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 *present* in *package*.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(setq sym (intern "CONFLICT")) → CONFLICT
(intern "CONFLICT" (make-package 'temp)) → TEMP::CONFLICT, NIL
(package-shadowing-symbols 'temp) → NIL
(shadowing-import sym 'temp) → T
(package-shadowing-symbols 'temp) → (CONFLICT)
```

Side Effects:

shadowing-import changes the state of the package system in such a way that the consistency rules do not hold across the change.

package's shadowing-symbols list is modified.

Affected By:

Current state of the package system.

See Also:

import, **unintern**, **package-shadowing-symbols**

delete-package *Function*

Syntax:

delete-package *package* → *boolean*

Arguments and Values:

package—a *package designator*.

boolean—a *boolean*.

Description:

delete-package deletes *package* from all package system data structures. If the operation is successful, **delete-package** returns true, otherwise **nil**. The effect of **delete-package** is that the name and nicknames of *package* cease to be recognized package names. The package *object* is still a *package* (*i.e.*, **packagep** is *true* of it) but **package-name** returns **nil**. The consequences of deleting the `COMMON-LISP` *package* or the `KEYWORD` *package* are undefined. The consequences of invoking any other package operation on *package* once it has been deleted are unspecified. In particular, the consequences of invoking **find-symbol**, **intern** and other functions that look for a

delete-package

symbol name in a *package* are unspecified if they are called with ***package*** bound to the deleted *package* or with the deleted *package* as an argument.

If *package* is a *package object* that has already been deleted, **delete-package** immediately returns **nil**.

After this operation completes, the *home package* of any *symbol* whose *home package* had previously been *package* is *implementation-dependent*. Except for this, *symbols accessible* in *package* are not modified in any other way; *symbols* whose *home package* is not *package* remain unchanged.

Examples:

```
(setq *foo-package* (make-package "FOO" :use nil))
(setq *foo-symbol*  (intern "FOO" *foo-package*))
(export *foo-symbol* *foo-package*)

(setq *bar-package* (make-package "BAR" :use '("FOO")))
(setq *bar-symbol*  (intern "BAR" *bar-package*))
(export *foo-symbol* *bar-package*)
(export *bar-symbol* *bar-package*)

(setq *baz-package* (make-package "BAZ" :use '("BAR")))

(symbol-package *foo-symbol*) → #<PACKAGE "FOO">
(symbol-package *bar-symbol*) → #<PACKAGE "BAR">

(prin1-to-string *foo-symbol*) → "FOO:FOO"
(prin1-to-string *bar-symbol*) → "BAR:BAR"

(find-symbol "FOO" *bar-package*) → FOO:FOO, :EXTERNAL

(find-symbol "FOO" *baz-package*) → FOO:FOO, :INHERITED
(find-symbol "BAR" *baz-package*) → BAR:BAR, :INHERITED

(packagep *foo-package*) → true
(packagep *bar-package*) → true
(packagep *baz-package*) → true

(package-name *foo-package*) → "FOO"
(package-name *bar-package*) → "BAR"
(package-name *baz-package*) → "BAZ"

(package-use-list *foo-package*) → ()
(package-use-list *bar-package*) → (#<PACKAGE "FOO">)
(package-use-list *baz-package*) → (#<PACKAGE "BAR">)
```

```
 (package-used-by-list *foo-package*) → (#<PACKAGE "BAR">)
 (package-used-by-list *bar-package*) → (#<PACKAGE "BAZ">)
 (package-used-by-list *baz-package*) → ()

 (delete-package *bar-package*)
▷ Error: Package BAZ uses package BAR.
▷ If continued, BAZ will be made to unuse-package BAR,
▷ and then BAR will be deleted.
▷ Type :CONTINUE to continue.
▷ Debug> :CONTINUE
→ T

 (symbol-package *foo-symbol*) → #<PACKAGE "FOO">
 (symbol-package *bar-symbol*) is unspecified

 (prin1-to-string *foo-symbol*) → "FOO:FOO"
 (prin1-to-string *bar-symbol*) is unspecified

 (find-symbol "FOO" *bar-package*) is unspecified

 (find-symbol "FOO" *baz-package*) → NIL, NIL
 (find-symbol "BAR" *baz-package*) → NIL, NIL

 (packagep *foo-package*) → T
 (packagep *bar-package*) → T
 (packagep *baz-package*) → T

 (package-name *foo-package*) → "FOO"
 (package-name *bar-package*) → NIL
 (package-name *baz-package*) → "BAZ"

 (package-use-list *foo-package*) → ()
 (package-use-list *bar-package*) is unspecified
 (package-use-list *baz-package*) → ()

 (package-used-by-list *foo-package*) → ()
 (package-used-by-list *bar-package*) is unspecified
 (package-used-by-list *baz-package*) → ()
```

Exceptional Situations:

If the *package designator* is a *name* that does not currently name a *package*, a *correctable* error of *type* **package-error** is signaled. If correction is attempted, no deletion action is attempted; instead, **delete-package** immediately returns **nil**.

If *package* is used by other *packages*, a *correctable* error of *type* **package-error** is signaled. If correction is attempted, **unuse-package** is effectively called to remove any dependencies, causing *package*'s *external symbols* to cease being *accessible* to those *packages* that use *package*. **delete-package** then deletes *package* just as it would have had there been no *packages* that used it.

See Also:

unuse-package

Notes:

make-package *Function*

Syntax:

make-package *package-name* &key *nicknames use* → *package*

Arguments and Values:

package-name—a *package name designator*.

nicknames—a *list* of *package name designators*. The default is the *empty list*.

use—a *list* of *package designators*. The default is *implementation-defined*.

package—a *package*.

Description:

Creates a new *package* with the name *package-name*.

Nicknames are additional *names* which may be used to refer to the new *package*.

use specifies zero or more *packages* the *external symbols* of which are to be inherited by the new *package*. See the *function* **use-package**.

Examples:

```
(make-package 'temporary :nicknames '("TEMP" "temp")) → #<PACKAGE "TEMPORARY">
(make-package "OWNER" :use '("temp")) → #<PACKAGE "OWNER">
(package-used-by-list 'temp) → (#<PACKAGE "OWNER">)
(package-use-list 'owner) → (#<PACKAGE "TEMPORARY">)
```

Affected By:

The existence of other *packages* in the system.

Exceptional Situations:

The consequences are unspecified if *packages* denoted by *use* do not exist.

A *correctable* error is signaled if the *package-name* or any of the *nicknames* is already the *name* or *nickname* of an existing *package*.

See Also:

defpackage, **use-package**

Notes:

In situations where the *packages* to be used contain symbols which would conflict, it is necessary to first create the package with `:use '()`, then to use **shadow** or **shadowing-import** to address the conflicts, and then after that to use **use-package** once the conflicts have been addressed.

When packages are being created as part of the static definition of a program rather than dynamically by the program, it is generally considered more stylistically appropriate to use **defpackage** rather than **make-package**.

with-package-iterator *Macro*

Syntax:

with-package-iterator (*name package-list-form* `&rest` *symbol-types*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

name—a *symbol*.

package-list-form—a *form*; evaluated once to produce a *package-list*.

package-list—a *designator* for a list of *package designators*.

symbol-type—one of the *symbols* `:internal`, `:external`, or `:inherited`.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

results—the *values* of the *forms*.

Description:

Within the lexical scope of the body *forms*, the *name* is defined via **macrolet** such that successive invocations of (*name*) will return the *symbols*, one by one, from the *packages* in *package-list*.

It is unspecified whether *symbols* inherited from multiple *packages* are returned more than once. The order of *symbols* returned does not necessarily reflect the order of *packages* in *package-list*.

with-package-iterator

When *package-list* has more than one element, it is unspecified whether duplicate *symbols* are returned once or more than once.

Symbol-types controls which *symbols* that are *accessible* in a *package* are returned as follows:

`:internal`

The *symbols* that are *present* in the *package*, but that are not *exported*.

`:external`

The *symbols* that are *present* in the *package* and are *exported*.

`:inherited`

The *symbols* that are *exported* by used *packages* and that are not *shadowed*.

When more than one argument is supplied for *symbol-types*, a *symbol* is returned if its *accessibility* matches any one of the *symbol-types* supplied. Implementations may extend this syntax by recognizing additional symbol accessibility types.

An invocation of (*name*) returns four values as follows:

1. A flag that indicates whether a *symbol* is returned (true means that a *symbol* is returned).

2. A *symbol* that is *accessible* in one the indicated *packages*.

3. The accessibility type for that *symbol*; *i.e.*, one of the symbols `:internal`, `:external`, or `:inherited`.

4. The *package* from which the *symbol* was obtained. The *package* is one of the *packages* present or named in *package-list*.

After all *symbols* have been returned by successive invocations of (*name*), then only one value is returned, namely **nil**.

The meaning of the second, third, and fourth *values* is that the returned *symbol* is *accessible* in the returned *package* in the way indicated by the second return value as follows:

`:internal`

Means *present* and not *exported*.

`:external`

Means *present* and *exported*.

`:inherited`

Means not *present* (thus not *shadowed*) but inherited from some used *package*.

It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the **with-package-iterator** form such as by returning some *closure* over the invocation *form*.

Any number of invocations of **with-package-iterator** can be nested, and the body of the innermost one can invoke all of the locally *established macros*, provided all those *macros* have distinct names.

Examples:

The following function should return **t** on any *package*, and signal an error if the usage of **with-package-iterator** does not agree with the corresponding usage of **do-symbols**.

```
(defun test-package-iterator (package)
  (unless (packagep package)
    (setq package (find-package package)))
  (let ((all-entries '())
        (generated-entries '()))
    (do-symbols (x package)
      (multiple-value-bind (symbol accessibility)
          (find-symbol (symbol-name x) package)
        (push (list symbol accessibility) all-entries)))
    (with-package-iterator (generator-fn package
                            :internal :external :inherited)
      (loop
        (multiple-value-bind (more? symbol pkg accessibility)
            (generator-fn)
          (unless more? (return))
          (let ((l (multiple-value-list (find-symbol (symbol-name symbol)
                                                     package))))
            (unless (equal l (list symbol accessibility))
              (error "Symbol ~S not found as ~S in package ~A [~S]"
                     symbol accessibility (package-name package) l))
            (push l generated-entries)))))
    (unless (and (subsetp all-entries generated-entries :test #'equal)
                 (subsetp generated-entries all-entries :test #'equal))
     (error "Generated entries and Do-Symbols entries don't correspond"))
    t))
```

The following function prints out every *present symbol* (possibly more than once):

```
(defun print-all-symbols ()
  (with-package-iterator (next-symbol (list-all-packages)
                          :internal :external)
```

```
(loop
  (multiple-value-bind (more? symbol) (next-symbol)
    (if more?
       (print symbol)
       (return))))))
```

Exceptional Situations:

with-package-iterator signals an error of *type* **program-error** if no *symbol-types* are supplied or if a *symbol-type* is not recognized by the implementation is supplied.

The consequences are undefined if the local function named *name* *established* by **with-package-iterator** is called after it has returned *false* as its *primary value*.

See Also:

Section 3.6 (Traversal Rules and Side Effects)

unexport *Function*

Syntax:

unexport *symbols* `&optional` *package* → t

Arguments and Values:

symbols—a *designator* for a *list* of *symbols*.

package—a *package designator*. The default is the *current package*.

Description:

unexport reverts external *symbols* in *package* to internal status; it undoes the effect of **export**.

unexport works only on *symbols present* in *package*, switching them back to internal status. If **unexport** is given a *symbol* that is already *accessible* as an *internal symbol* in *package*, it does nothing.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(export (intern "CONTRABAND" (make-package 'temp)) 'temp) → T
(find-symbol "CONTRABAND") → NIL, NIL
(use-package 'temp) → T
(find-symbol "CONTRABAND") → CONTRABAND, :INHERITED
(unexport 'contraband 'temp) → T
(find-symbol "CONTRABAND") → NIL, NIL
```

Side Effects:

Package system is modified.

Affected By:

Current state of the package system.

Exceptional Situations:

If **unexport** is given a *symbol* not *accessible* in *package* at all, an error of *type* **package-error** is signaled.

The consequences are undefined if *package* is the `KEYWORD` *package* or the `COMMON-LISP` *package*.

See Also:

export, Section 11.1 (Package Concepts)

unintern *Function*

Syntax:

unintern *symbol* `&optional` *package* → *boolean*

Arguments and Values:

symbol—a *symbol*.

package—a *package designator*. The default is the *current package*.

boolean—a *boolean*.

Description:

unintern removes *symbol* from *package*. If *symbol* is *present* in *package*, it is removed from *package* and also from *package*'s *shadowing symbols list* if it is present there. If *package* is the *home package* for *symbol*, *symbol* is made to have no *home package*. *Symbol* may continue to be *accessible* in *package* by inheritance.

Use of **unintern** can result in a *symbol* that has no recorded *home package*, but that in fact is *accessible* in some *package*. Common Lisp does not check for this pathological case, and such *symbols* are always printed preceded by `#:`.

unintern returns *true* if it removes *symbol*, and **nil** otherwise.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(setq temps-unpack (intern "UNPACK" (make-package 'temp))) → TEMP::UNPACK
(unintern temps-unpack 'temp) → T
```

```
(find-symbol "UNPACK" 'temp) → NIL, NIL
temps-unpack → #:UNPACK
```

Side Effects:

unintern changes the state of the package system in such a way that the consistency rules do not hold across the change.

Affected By:

Current state of the package system.

Exceptional Situations:

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.

See Also:

Section 11.1 (Package Concepts)

in-package *Macro*

Syntax:

in-package *name* → *package*

Arguments and Values:

name—a *package name designator*; not evaluated.

package—the *package* named by *name*.

Description:

Causes the the *package* named by *name* to become the *current package*—that is, the *value* of ***package***. If no such *package* already exists, an error of *type* **package-error** is signaled.

Everything **in-package** does is also performed at compile time if the call appears as a *top level form*.

Side Effects:

The *variable* ***package*** is assigned. If the **in-package** *form* is a *top level form*, this assignment also occurs at compile time.

Exceptional Situations:

An error of *type* **package-error** is signaled if the specified *package* does not exist.

See Also:

package

Notes:

unuse-package *Function*

Syntax:

unuse-package *packages-to-unuse* &optional *package* → t

Arguments and Values:

packages-to-unuse—a *designator* for a *list* of *package designators*.

package—a *package designator*. The default is the *current package*.

Description:

unuse-package causes *package* to cease inheriting all the *external symbols* of *packages-to-unuse*; **unuse-package** undoes the effects of **use-package**. The *packages-to-unuse* are removed from the *use list* of *package*.

Any *symbols* that have been *imported* into *package* continue to be *present* in *package*.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(export (intern "SHOES" (make-package 'temp)) 'temp) → T
(find-symbol "SHOES") → NIL, NIL
(use-package 'temp) → T
(find-symbol "SHOES") → SHOES, :INHERITED
(find (find-package 'temp) (package-use-list 'common-lisp-user)) → #<PACKAGE "TEMP">
(unuse-package 'temp) → T
(find-symbol "SHOES") → NIL, NIL
```

Side Effects:

The *use list* of *package* is modified.

Affected By:

Current state of the package system.

See Also:

use-package, **package-use-list**

use-package *Function*

Syntax:

use-package *packages-to-use* &optional *package* → t

Arguments and Values:

packages-to-use—a *designator* for a *list* of *package designators*. The KEYWORD *package* may not be supplied.

package—a *package designator*. The KEYWORD *package* cannot be supplied. The default is the *current package*.

Description:

use-package causes *package* to inherit all the *external symbols* of *packages-to-use*. The inherited *symbols* become *accessible* as *internal symbols* of *package*.

Packages-to-use are added to the *use list* of *package* if they are not there already. All *external symbols* in *packages-to-use* become *accessible* in *package* as *internal symbols*. **use-package** does not cause any new *symbols* to be *present* in *package* but only makes them *accessible* by inheritance.

use-package checks for name conflicts between the newly imported symbols and those already *accessible* in *package*. A name conflict in **use-package** between two external symbols inherited by *package* from *packages-to-use* may be resolved in favor of either *symbol* by *importing* one of them into *package* and making it a shadowing symbol.

Examples:

```
(export (intern "LAND-FILL" (make-package 'trash)) 'trash) → T
(find-symbol "LAND-FILL" (make-package 'temp)) → NIL, NIL
(package-use-list 'temp) → (#<PACKAGE "TEMP">)
(use-package 'trash 'temp) → T
(package-use-list 'temp) → (#<PACKAGE "TEMP"> #<PACKAGE "TRASH">)
(find-symbol "LAND-FILL" 'temp) → TRASH:LAND-FILL, :INHERITED
```

Side Effects:

The *use list* of *package* may be modified.

See Also:

unuse-package, **package-use-list**, Section 11.1 (Package Concepts)

Notes:

It is permissible for a *package* P_1 to *use* a *package* P_2 even if P_2 already uses P_1. The using of *packages* is not transitive, so no problem results from the apparent circularity.

defpackage *Macro*

Syntax:

defpackage *defined-package-name* ⟦↓*option*⟧ → *package*

option::={(`:nicknames` {*nickname*}*)}* |
(`:documentation` *string*) |
{(`:use` {*package-name*}*)}* |
{(`:shadow` {↓*symbol-name*}*)}* |
{(`:shadowing-import-from` *package-name* {↓*symbol-name*}*)}* |
{(`:import-from` *package-name* {↓*symbol-name*}*)}* |
{(`:export` {↓*symbol-name*}*)}* |
{(`:intern` {↓*symbol-name*}*)}* |
(`:size` *integer*)

symbol-name::=(*symbol* | *string*)

Arguments and Values:

defined-package-name—a *package name designator*.

package-name—a *package designator*.

nickname—a *package name designator*.

symbol-name—a *symbol name designator*.

package—the *package* named *package-name*.

Description:

defpackage creates a *package* as specified and returns the *package*.

If *defined-package-name* already refers to an existing *package*, the name-to-package mapping for that name is not changed. If the new definition is at variance with the current state of that *package*, the consequences are undefined; an implementation might choose to modify the existing *package* to reflect the new definition. If *defined-package-name* is a *symbol*, its *name* is used.

The standard *options* are described below.

`:nicknames`

The arguments to `:nicknames` set the *package*'s nicknames to the supplied names.

`:documentation`

The argument to `:documentation` specifies a *documentation string*; it is attached as a

defpackage

documentation string to the *package*. At most one `:documentation` option can appear in a single **defpackage** *form*.

`:use`

The arguments to `:use` set the *packages* that the *package* named by *package-name* will inherit from. If `:use` is not supplied, it defaults to the same *implementation-dependent* value as the `:use` *argument* to **make-package**.

`:shadow`

The arguments to `:shadow`, *symbol-names*, name *symbols* that are to be created in the *package* being defined. These *symbols* are added to the list of shadowing *symbols* effectively as if by **shadow**.

`:shadowing-import-from`

The *symbols* named by the argument *symbol-names* are found (involving a lookup as if by **find-symbol**) in the specified *package-name*. The resulting *symbols* are *imported* into the *package* being defined, and placed on the shadowing symbols list as if by **shadowing-import**. In no case are *symbols* created in any *package* other than the one being defined.

`:import-from`

The *symbols* named by the argument *symbol-names* are found in the *package* named by *package-name* and they are *imported* into the *package* being defined. In no case are *symbols* created in any *package* other than the one being defined.

`:export`

The *symbols* named by the argument *symbol-names* are found or created in the *package* being defined and *exported*. The `:export` option interacts with the `:use` option, since inherited *symbols* can be used rather than new ones created. The `:export` option interacts with the `:import-from` and `:shadowing-import-from` options, since *imported* symbols can be used rather than new ones created. If an argument to the `:export` option is *accessible* as an (inherited) *internal symbol* via **use-package**, that the *symbol* named by *symbol-name* is first *imported* into the *package* being defined, and is then *exported* from that *package*.

`:intern`

The *symbols* named by the argument *symbol-names* are found or created in the *package* being defined. The `:intern` option interacts with the `:use` option, since inherited *symbols* can be used rather than new ones created.

defpackage

`:size`

The argument to the `:size` option declares the approximate number of *symbols* expected in the *package*. This is an efficiency hint only and might be ignored by an implementation.

The order in which the options appear in a **defpackage** form is irrelevant. The order in which they are executed is as follows:

1. `:shadow` and `:shadowing-import-from`.

2. `:use`.

3. `:import-from` and `:intern`.

4. `:export`.

Shadows are established first, since they might be necessary to block spurious name conflicts when the `:use` option is processed. The `:use` option is executed next so that `:intern` and `:export` options can refer to normally inherited *symbols*. The `:export` option is executed last so that it can refer to *symbols* created by any of the other options; in particular, *shadowing symbols* and *imported symbols* can be made external.

If a defpackage *form* appears as a *top level form*, all of the actions normally performed by this *macro* at load time must also be performed at compile time.

Examples:

```
(defpackage "MY-PACKAGE"
  (:nicknames "MYPKG" "MY-PKG")
  (:use "COMMON-LISP")
  (:shadow "CAR" "CDR")
  (:shadowing-import-from "VENDOR-COMMON-LISP"  "CONS")
  (:import-from "VENDOR-COMMON-LISP"  "GC")
  (:export "EQ" "CONS" "FROBOLA")
  )


(defpackage my-package
  (:nicknames mypkg :MY-PKG)  ; remember Common Lisp conventions for case
  (:use common-lisp)          ; conversion on symbols
  (:shadow CAR :cdr #:cons)
  (:export "CONS")            ; this is the shadowed one.
  )
```

defpackage

Affected By:

Existing *packages*.

Exceptional Situations:

If one of the supplied `:nicknames` already refers to an existing *package*, an error of *type* **package-error** is signaled.

An error of *type* **program-error** should be signaled if `:size` or `:documentation` appears more than once.

Since *implementations* might allow extended *options* an error of *type* **program-error** should be signaled if an *option* is present that is not actually supported in the host *implementation*.

The collection of *symbol-name* arguments given to the options `:shadow`, `:intern`, `:import-from`, and `:shadowing-import-from` must all be disjoint; additionally, the *symbol-name* arguments given to `:export` and `:intern` must be disjoint. Disjoint in this context is defined as no two of the *symbol-names* being **string=** with each other. If either condition is violated, an error of *type* **program-error** should be signaled.

For the `:shadowing-import-from` and `:import-from` options, a *correctable error* of *type* **package-error** is signaled if no *symbol* is *accessible* in the *package* named by *package-name* for one of the argument *symbol-names*.

Name conflict errors are handled by the underlying calls to **make-package**, **use-package**, **import**, and **export**. See Section 11.1 (Package Concepts).

See Also:

documentation, Section 11.1 (Package Concepts), Section 3.2 (Compilation)

Notes:

The `:intern` option is useful if an `:import-from` or a `:shadowing-import-from` option in a subsequent call to **defpackage** (for some other *package*) expects to find these *symbols accessible* but not necessarily external.

It is recommended that the entire *package* definition is put in a single place, and that all the *package* definitions of a program are in a single file. This file can be *loaded* before *loading* or compiling anything else that depends on those *packages*. Such a file can be read in the `COMMON-LISP-USER` *package*, avoiding any initial state issues.

defpackage cannot be used to create two "mutually recursive" packages, such as:

```
(defpackage my-package
  (:use common-lisp your-package)    ;requires your-package to exist first
  (:export "MY-FUN"))
(defpackage your-package
  (:use common-lisp)
  (:import-from my-package "MY-FUN") ;requires my-package to exist first
```

```
(:export "MY-FUN"))
```

However, nothing prevents the user from using the *package*-affecting functions such as **use-package**, **import**, and **export** to establish such links after a more standard use of **defpackage**.

The macroexpansion of **defpackage** could usefully canonicalize the names into *strings*, so that even if a source file has random *symbols* in the **defpackage** form, the compiled file would only contain *strings*.

Frequently additional *implementation-dependent* options take the form of a *keyword* standing by itself as an abbreviation for a list `(keyword T)`; this syntax should be properly reported as an unrecognized option in implementations that do not support it.

do-symbols, do-external-symbols, do-all-symbols

Macro

Syntax:

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

→ {*result*}*

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

→ {*result*}*

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

→ {*result*}*

Arguments and Values:

var—a *variable name*; not evaluated.

package—a *package designator*; evaluated. The default in **do-symbols** and **do-external-symbols** is the *current package*.

result-form—a *form*; evaluated as described below. The default is **nil**.

declaration—a **declare** *expression*; not evaluated.

tag—a *go tag*; not evaluated.

statement—a *compound form*; evaluated as described below.

do-symbols, do-external-symbols, do-all-symbols

results—the *values* returned by the *result-form* if a *normal return* occurs, or else, if an *explicit return* occurs, the *values* that were transferred.

Description:

do-symbols, **do-external-symbols**, and **do-all-symbols** iterate over the *symbols* of *packages*. For each *symbol* in the set of *packages* chosen, the *var* is bound to the *symbol*, and the *statements* in the body are executed. When all the *symbols* have been processed, *result-form* is evaluated and returned as the value of the macro.

do-symbols iterates over the *symbols accessible* in *package*. *Statements* may execute more than once for *symbols* that are inherited from multiple *packages*.

do-all-symbols iterates on every *registered package*. **do-all-symbols** will not process every *symbol* whatsoever, because a *symbol* not *accessible* in any *registered package* will not be processed. **do-all-symbols** may cause a *symbol* that is *present* in several *packages* to be processed more than once.

do-external-symbols iterates on the external symbols of *package*.

When *result-form* is evaluated, *var* is bound and has the value **nil**.

An *implicit block* named **nil** surrounds the entire **do-symbols**, **do-external-symbols**, or **do-all-symbols** *form*. **return** or **return-from** may be used to terminate the iteration prematurely.

If execution of the body affects which *symbols* are contained in the set of *packages* over which iteration is occurring, other than to remove the *symbol* currently the value of *var* by using **unintern**, the consequences are undefined.

For each of these macros, the *scope* of the name binding does not include any initial value form, but the optional result forms are included.

Any *tag* in the body is treated as with **tagbody**.

Examples:

```
(make-package 'temp :use nil) → #<PACKAGE "TEMP">
(intern "SHY" 'temp) → TEMP::SHY, NIL ;SHY will be an internal symbol
                                      ;in the package TEMP
(export (intern "BOLD" 'temp) 'temp)  → T  ;BOLD will be external
(let ((lst ()))
  (do-symbols (s (find-package 'temp)) (push s lst))
  lst)
→ (TEMP::SHY TEMP:BOLD)
or→ (TEMP:BOLD TEMP::SHY)
(let ((lst ()))
  (do-external-symbols (s (find-package 'temp) lst) (push s lst))
```

```
   lst)
→ (TEMP:BOLD)
 (let ((lst ()))
   (do-all-symbols (s lst)
     (when (eq (find-package 'temp) (symbol-package s)) (push s lst)))
   lst)
→ (TEMP::SHY TEMP:BOLD)
or
→ (TEMP:BOLD TEMP::SHY)
```

See Also:

intern, **export**, Section 3.6 (Traversal Rules and Side Effects)

intern *Function*

Syntax:

intern *string* &optional *package* → *symbol, status*

Arguments and Values:

string—a *string*.

package—a *package designator*. The default is the *current package*.

symbol—a *symbol*.

status—one of :inherited, :external, :internal, or **nil**.

Description:

intern enters a *symbol* named *string* into *package*. If a *symbol* whose name is the same as *string* is already *accessible* in *package*, it is returned. If no such *symbol* is *accessible* in *package*, a new *symbol* with the given name is created and entered into *package* as an *internal symbol*, or as an *external symbol* if the *package* is the KEYWORD *package*; *package* becomes the *home package* of the created *symbol*.

The first value returned by **intern**, *symbol*, is the *symbol* that was found or created. The meaning of the *secondary value*, *status*, is as follows:

:internal

The *symbol* was found and is *present* in *package* as an *internal symbol*.

:external

The *symbol* was found and is *present* as an *external symbol*.

`:inherited`

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

nil

No pre-existing *symbol* was found, so one was created.

It is *implementation-dependent* whether the *string* that becomes the new *symbol*'s *name* is the given *string* or a copy of it. Once a *string* has been given as the *string argument* to *intern* in this situation where a new *symbol* is created, the consequences are undefined if a subsequent attempt is made to alter that *string*.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(intern "Never-Before") → |Never-Before|, NIL
(intern "Never-Before") → |Never-Before|, :INTERNAL
(intern "NEVER-BEFORE" "KEYWORD") → :NEVER-BEFORE, NIL
(intern "NEVER-BEFORE" "KEYWORD") → :NEVER-BEFORE, :EXTERNAL
```

See Also:

find-symbol, **read**, **symbol**, **unintern**, Section 2.3.4 (Symbols as Tokens)

Notes:

intern 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.

package-name *Function*

Syntax:

package-name *package* → *name*

Arguments and Values:

package—a *package designator*.

name—a *string* or **nil**.

Description:

package-name returns the *string* that names *package*, or **nil** if the *package designator* is a *package object* that has no name (see the *function* **delete-package**).

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(package-name *package*) → "COMMON-LISP-USER"
(package-name (symbol-package :test)) → "KEYWORD"
(package-name (find-package 'common-lisp)) → "COMMON-LISP"

(defvar *foo-package* (make-package "FOO"))
(rename-package "FOO" "FOO0")
(package-name *foo-package*) → "FOO0"
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *package* is not a *package designator*.

package-nicknames *Function*

Syntax:

package-nicknames *package* → *nicknames*

Arguments and Values:

package—a *package designator*.

nicknames—a *list* of *strings*.

Description:

Returns the *list* of nickname *strings* for *package*, not including the name of *package*.

Examples:

```
 (package-nicknames (make-package 'temporary
                                   :nicknames '("TEMP" "temp")))
→ ("temp" "TEMP")
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *package* is not a *package designator*.

package-shadowing-symbols *Function*

Syntax:

package-shadowing-symbols *package* → *symbols*

Arguments and Values:

package—a *package designator*.

symbols—a *list* of *symbols*.

Description:

Returns a *list* of *symbols* that have been declared as *shadowing symbols* in *package* by **shadow** or **shadowing-import** (or the equivalent **defpackage** options). All *symbols* on this *list* are *present* in *package*.

Examples:

```
(package-shadowing-symbols (make-package 'temp)) → ()
(shadow 'cdr 'temp) → T
(package-shadowing-symbols 'temp) → (TEMP::CDR)
(intern "PILL" 'temp) → TEMP::PILL, NIL
(shadowing-import 'pill 'temp) → T
(package-shadowing-symbols 'temp) → (PILL TEMP::CDR)
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *package* is not a *package designator*.

See Also:

shadow, **shadowing-import**

Notes:

Whether the list of *symbols* is *fresh* is *implementation-dependent*.

package-use-list *Function*

Syntax:

package-use-list *package* → *use-list*

Arguments and Values:

package—a *package designator*.

use-list—a *list* of *package objects*.

Description:

Returns a *list* of other *packages* used by *package*.

Examples:

```
(package-use-list (make-package 'temp)) → (#<PACKAGE "COMMON-LISP">)
(use-package 'common-lisp-user 'temp) → T
(package-use-list 'temp) → (#<PACKAGE "COMMON-LISP"> #<PACKAGE "COMMON-LISP-USER">)
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *package* is not a *package designator*.

See Also:

use-package, **unuse-package**

package-used-by-list *Function*

Syntax:

package-used-by-list *package* → *used-by-list*

Arguments and Values:

package—a *package designator*.

used-by-list—a *list* of *package objects*.

Description:

package-used-by-list returns a *list* of other *packages* that use *package*.

Examples:

```
(package-used-by-list (make-package 'temp)) → ()
(make-package 'trash :use '(temp)) → #<PACKAGE "TRASH">
(package-used-by-list 'temp) → (#<PACKAGE "TRASH">)
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *package* is not a *package*.

See Also:

use-package, **unuse-package**

packagep *Function*

Syntax:

packagep *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is of *type* **package**; otherwise, returns *false*.

Examples:

```
(packagep *package*) → true
(packagep 'common-lisp) → false
(packagep (find-package 'common-lisp)) → true
```

Notes:

```
(packagep object) ≡ (typep object 'package)
```

package *Variable*

Value Type:

a *package object*.

Initial Value:

the COMMON-LISP-USER *package*.

Description:

Whatever *package object* is currently the *value* of ***package*** is referred to as the *current package*.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
*package* → #<PACKAGE "COMMON-LISP-USER">
(make-package "SAMPLE-PACKAGE" :use '("COMMON-LISP"))
→ #<PACKAGE "SAMPLE-PACKAGE">
(list
```

```
   (symbol-package
     (let ((*package* (find-package 'sample-package)))
       (setq *some-symbol* (read-from-string "just-testing"))))
   *package*)
→ (#<PACKAGE "SAMPLE-PACKAGE"> #<PACKAGE "COMMON-LISP-USER">)
 (list (symbol-package (read-from-string "just-testing"))
       *package*)
→ (#<PACKAGE "COMMON-LISP-USER"> #<PACKAGE "COMMON-LISP-USER">)
 (eq 'foo (intern "FOO")) → true
 (eq 'foo (let ((*package* (find-package 'sample-package)))
            (intern "FOO")))
→ false
```

Affected By:

load, **compile-file**, **in-package**

See Also:

compile-file, **in-package**, **load**, **package**

package-error *Condition Type*

Class Precedence List:

package-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **package-error** consists of *error conditions* related to operations on *packages*. The offending *package* (or *package name*) is initialized by the `:package` initialization argument to **make-condition**, and is *accessed* by the *function* **package-error-package**.

See Also:

package-error-package, Chapter 9 (Conditions)

package-error-package

package-error-package *Function*

Syntax:

package-error-package *condition* → *package*

Arguments and Values:

condition—a *condition* of *type* **package-error**.

package—a *package designator*.

Description:

Returns a *designator* for the offending *package* in the *situation* represented by the *condition*.

Examples:

```
 (package-error-package
   (make-condition 'package-error
     :package (find-package "COMMON-LISP")))
→ #<Package "COMMON-LISP">
```

See Also:

package-error

Table of Contents