27. Packages

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

The Lisp Machine consists of a huge Lisp environment, in which many programs must coexist. All of the operating system, the compiler, the editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program that you use during a session must be loaded into the same environment. Each of these programs is composed of a group of functions; apparently each function must have its own distinct name to avoid conflicts. For example, if the compiler had a function named **pull**, and you loaded a program which had its own function named **pull**, the compiler's **pull** would be redefined, probably breaking the compiler.

It would not really be possible to prevent these conflicts, since the programs are written by many different people who could never get together to hash out who gets the privilege of using a specific name such as **pull**.

Now, if we are to enable two programs to coexist in the Lisp world, each with its own function **pull**, then each program must have its own symbol named **pull**, because there can't be two function definitions on the same symbol. This means that separate *name spaces*—mappings between names and symbols—must be provided for the two programs. The package system is designed to do just that.

Under the package system, the author of a program or a group of closely related programs identifies them together as a *package*. The package system associates a distinct name space with each package.

Here is an example: suppose there are two programs named **chaos** and **arpa**, for handling the Chaosnet and Arpanet respectively. The author of each program wants to have a function called **get-packet**, which reads in a packet from the network (or something). Also, each wants to have a function called **allocate-pbuf**, which allocates the packet buffer. Each "get" routine first allocates a packet buffer, and then reads bits into the buffer; therefore, each version of **get-packet** should call the respective version of **allocate-pbuf**.

Without the package system, the two programs could not coexist in the same Lisp environment. But the package feature can be used to provide a separate name space for each program. What is required is to define a package named **chaos** to contain the Chaosnet program, and another package **arpa** to hold the Arpanet program. When the Chaosnet program is read into the machine, its symbols would be entered in the **chaos** package's name space. So when the Chaosnet program's **get-packet** referred to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** name space would be found, which would be the **allocate-pbuf** of the Chaosnet program—the right one. Similarly, the Arpanet program's **get-packet** would be read in using the **arpa** package and would refer to the Arpanet program's **allocate-pbuf**.

In order to have multiple name spaces, the function **intern**, which searches for a name, must allow the name space to be specified. **intern** accepts an optional second argument which is the package to search.

It's obvious that every file has to be loaded into the right package to serve its purpose. It may not be so obvious that every file must be compiled in the right package, but it's just as true. Luckily, this usually happens automatically.

The system can get the package of a source file from its -*- line. For instance, you can put at the front of your file a line such as

```
        ; -*- Mode:Lisp; Package:System-Internals -*-
```

The compiler puts the package name into the QFASL file for use when it is loaded. If a file doesn't have such a package specification in it, the system loads it into the current package and tells you what it did.

27.1 The Current Package

At any time, one package is the *current package*. By default, symbol lookup happens in the current package.

package *Variable*
package *Variable*

The value of the this variable is the current package. **intern** searches this package if it is not given a second argument. Many other functions for operating on packages also use this as the default.

Setting or binding the variable changes the current package. May the Goddess help you if you set it to something that isn't a package!

The two names are synonymous.

Each process or stack group can have its own setting for the current package by binding ***package*** with **let**. The actual current package at any time is the value bound by the process which is running. The bindings of another process are irrelevant until the process runs.

pkg-bind *pkg body...* *Macro*

pkg may be a package or a package name. The forms of the *body* are evaluated sequentially with the variable ***package*** bound to the package named by *pkg*.
Example:

```
        (pkg-bind "ZWEI"
          (read-from-string function-name))
```

When a file is loaded, ***package*** is bound to the correct package for the file (the one named in the file's -*- line). The Chaosnet program file has **Package: Chaos;** in the -*- line, and therefore its symbols are looked up in the **chaos** package. A QFASL file has an encoded representation of the -*- line of the source file; it looks different, but it serves the same purpose.

The current package is also relevant when you type Lisp expressions on the keyboard; it controls the reading of the symbols that you type. Initially it is the package **user**. You can select a different package using **pkg-goto**, or even by **setqing *package***. If you are working with the Chaosnet program, it might be useful to type **(pkg-goto 'chaos)** so that your symbols are found in the **chaos** package by default. The Lisp listen loop binds ***package*** so that **pkg-goto** in one Lisp listener does not affect others, or any other processes whatever.

pkg-goto *package* &optional *globally*

Sets *package* to *package*. if *package* is suitable. (Autoexporting packages used by other packages are not suitable because it you could cause great troubles by interning new symbols in them). *package* may be specified as a package object or the name of one. If globally is non-nil, then this function also calls pkg-goto-globally (see below)

The Zmacs editor records the correct package for each buffer; it is determined from the file's -*- line. This package is used whenever expressions are read from the buffer. So if you edit the definition of the Chaosnet get-packet and recompile it, the new definition is read in the chaos package. The current buffer's package is also used for all expressions or symbols typed by the user. Thus, if you type Meta-. allocate-pbuf while looking at the Chaosnet program, you get the definition of the allocate-pbuf function in the chaos package.

The variable *package* also has a global binding, which is in effect in any process or stack group which does not rebind the variable. New processes that do bind *package* generally use the global binding to initialize their own bindings, doing (let ((*package* *package*)) ...). Therefore, it can be useful to set the global binding. But you cannot do this with setq or pkg-goto from a Lisp listener, or in a file, because that will set the local binding of *package* instead. Therefore you must use setq-globally (page 35) or pkg-goto-globally.

pkg-goto-globally *package*

Sets the global binding of *package* to *package*. An error is signaled if *package* is not suitable. Bindings of *package* other than the the global one are not changed, including the current binding if it is not the global one.

The name of the current package is always displayed in the middle of the who line, with a colon following it. This describes the process which the who line in general is describing; normally, the process of the selected window. No matter how the current package is changed, the who line will eventually show it (at one-second intervals). Thus, while a file is being loaded, the who line displays that file's package; in the editor, the who line displays the package of the selected buffer.

27.2 Package Prefixes

The separation of name spaces is not an uncrossable gulf. Consider a program for accessing files, using the Chaosnet. It may be useful to put it in a distinct package file-access, not chaos, so that the programs are protected from accidental name conflicts. But the file program cannot exist without referring to the functions of the Chaosnet program.

The colon character (':') has a special meaning to the Lisp reader. When the reader sees a colon preceded by the name of a package, it reads the next Lisp object with *package* bound to that package. Thus, to refer to the symbol connect in package chaos, we write chaos:connect. Some symbols documented in this manual require package prefixes to refer to them; they are always written with an appropriate prefix.

Similarly, if the chaos program wanted to refer to the arpa program's allocate-pbuf function (for some reason), it could use arpa:allocate-pbuf.

Package prefixes are printed on output also. If you would need a package prefix to refer to a symbol on input, then the symbol is printed with a suitable package prefix if it supposed to be printed readably (prin1, as opposed to princ). Just as the current package affects how a symbol is read, it also affects how the symbol is printed. A symbol available in the current package is never printed with a package prefix.

The printing of package prefixes makes it possible to print list structure containing symbols from many packages and read the text to produce an equal list with the same symbols in it—provided the current package when the text is read is the same one that was current when the text was printed.

The package name in a package prefix is read just like a symbol name. This means that escape characters can be used to include special characters in the package name. Thus, foo/:bar:test refers to the symbol test in the package whose name is "FOO:BAR", and so does |FOO:BAR|:test. Also, letters are converted to upper case unless they are escaped. For this reason, the actual name of a package is normally all upper case, but you can use either case when you write a package prefix.

In Common Lisp programs, simple colon prefixes are supposed to be used only for referring to external symbols (see page 642). To refer to other symbols, one is supposed to use two colons, as in chaos::lose-it-later. The Lisp machine tradition is to allow reference to any symbol with a single colon. Since this is upward compatible with what is allowed in Common Lisp, single-colon references are always allowed. However, double-colon prefixes are printed for internal symbols when Common Lisp syntax is in use, so that data printed on a Lisp Machine can be read by other Common Lisp implementations.

27.3 Home Packages of Symbols

Each symbol remembers one package which it belongs to: normally, the first one it was ever interned in. This package is available as (symbol-package *symbol*).

With make-symbol (see page 133) it is possible to create a symbol that has never been interned in any package. It is called an *uninterned symbol*, and it remains one as long as nobody interns it. The package cell of an uninterned symbol contains nil. Uninterned symbols print with #: as a prefix, as in #:foo. This syntax can be used as input to create an uninterned symbol with a specific name; but a new symbol is created each time you type it, since the mechanism which normally makes symbols unique is interning in a package. Thus, (eq #:foo #:foo) returns nil.

symbol-package *symbol*

Returns the contents of *symbol*'s package cell, which is the package which owns *symbol*, or nil if *symbol* is uninterned.

package-cell-location *symbol*

Returns a locative pointer to *symbol*'s package cell. It is preferable to write

```
(locf (symbol-package symbol))
```

rather than calling this function explicitly.

Printing of package prefixes is based on the contents of the symbol's package cell. If the cell contains the **chaos** package, then **chaos:** is printed as the prefix when a prefix is necessary. As a result of obscure actions involving interning and uninterning in multiple packages, the symbol may not actually be present in **chaos** any more. Then the printed prefix is inaccurate. This cannot be helped. If the symbol is not where it claims to be, there is no easy way to find wherever it might be.

27.4 Keywords

Distinct name spaces are useful for symbols which have function definitions or values, to enable them to be used independently by different programs.

Another way to use a symbol is to check for it with **eq**. Then there is no possibility of name conflict. For example, the function **open**, part of the file system, checks for the symbol **:error** in its input using **eq**. A user function might do the same thing. Then the symbol **:error** is meaningful in two contexts, but these meanings do not affect each other. The fact that a user program contains the code **(eq sym :error)** does not interfere with the function of system code which contains a similar expression.

There is no need to separate name spaces for symbols used in this way. In fact, it would be a disadvantage. If both the Chaosnet program and the Arpanet program wish to recognize a keyword named "address", for similar purposes (naturally), it is very useful for programs that can call either one if it is the *same* keyword for either program. But which should it be? **chaos:address? arpa:address?**

To avoid this uncertainty, one package called **keyword** has been set aside for the keywords of all programs. The Chaosnet and Arpanet programs would both look for **keyword:address**, normally written as just **:address.**

Symbols in **keyword** are the normal choice for names of keyword arguments; if you use **&key** to process them, code is automatically generated to look for for symbols in **keyword**. They are also the normal choice for flavor operation names, and for any set of named options meaningful in a specific context.

keyword and the symbols belonging to it are treated differently from other packages in a couple of ways designed to make them more convenient for this usage.

* Symbols belonging to **keyword** are constants; they always evaluate to themselves. (This is brought about by storing the symbol in its own value cell when the symbol is placed in the package). So you can write just **:error** rather than **':error**. The nature of the application of keywords is such that they would always be quoted if they were not constant.

* A colon by itself is a sufficient package prefix for **keyword**. This is because keywords are the most frequent application of package prefixes.

keywordp *object*

t if *object* is a symbol which belongs to the keyword package.

There are certain cases when a keyword should *not* be used for a symbol to be checked for with eq. Usually this is when the symbol 1) does not need to be known outside of a single program, and 2) is to be placed in shared data bases such as property lists of symbols which may sometimes be in global or keyword. For example, if the Chaosnet program were to record the existence of a host named CAR by placing an :address property on the symbol :car, or the symbol car (notice that chaos:car *is* car), it would risk conflicts with other programs that might wish to use the :address property of symbols in general. It is better to call the property chaos:address.

27.5 Inheritance between Name Spaces

In the simplest (but not the default) case, a package is independent of all other packages. This is not the default because it is not usually useful. Consider the standard Lisp function and variables names, such as car: how can the Chaosnet program, using the chaos package, access them? One way would be to install all of them in the chaos package, and every other package. But it is better to have one table of the standard Lisp symbols and refer to it where necessary. This is called *inheritance*. The single package global is the only one which actually contains the standard Lisp symbols; other packages such as chaos contain directions to "search global too".

Each package has a hash table of the symbols. The symbols in this table are said to be *present* (more explicitly, *present directly*) in the package, or *interned* in it. In addition, each package has a list of other packages to inherit from. By default, this list contains the package global and no others; but packages can be added and removed at any time with the functions use-package and unuse-package. We say that a package *uses* the packages it inherits from. Both the symbols present directly in the package and the symbols it inherits are said to be *available* in the package.

Here's how this works in the above example. When the Chaosnet program is read into the Lisp world, the current package would be the chaos package. Thus all of the symbols in the Chaosnet program would be interned in the chaos package. If there is a reference to a standard Lisp symbol such as append, nothing is found in the chaos package's own table; no symbol of that name is present directly in chaos. Therefore the packages used by chaos are searched, including global. Since global contains a symbol named append, that symbol is found. If, however, there is a reference to a symbol that is not standard, such as get-packet, the first time it is used it is not found in either chaos or global. So intern makes a new symbol named get-packet, and installs it in the chaos package. When get-packet is referred to later in the Chaosnet program, intern finds get-packet immediately in the chaos package. global does not need to be searched.

When the Arpanet program is read in, the current package is arpa instead of chaos. When the Arpanet program refers to append, it gets the global one; that is, it shares the same one that the Chaosnet program got. However, if it refers to get-packet, it does *not* get the same one the Chaosnet program got, because the chaos package is presumably not used by arpa. The get-packet in chaos not being available, no symbol is found, so a new one is created and placed in the arpa package. Further references in the Arpanet program find that get-packet.

This is the desired result: the packages share the standard Lisp symbols only.

Inheritance between other packages can also be useful, but it must be restricted: inheriting only some of the symbols of the used package. If the file access program refers frequently to the advertised symbols of the Chaosnet program—the connection states, such as **open-state**, functions such as **connect**, **listen** and **open-stream**, and others—it might be convenient to be able to refer to these symbols from the **file-access** package without need for package prefixes.

One way to do this is to place the appropriate symbols of the **chaos** package into the **file-access** package as well. Then they can be accessed by the file access program just like its own symbols. Such sharing of symbols between packages never happens from the ordinary operation of packages, but it can be requested explicitly using **import**.

import *symbols* &optional (*package* ***package***)

Is the standard Common Lisp way to insert a specific symbol or symbols into a package. *symbols* is a symbol or a list of symbols. Each of the specified symbols becomes present directly in *package*.

If a symbol with the same name is already present (directly or by inheritance) in *package*, an error is signaled. On proceeding, you can say whether to leave the old symbol there or replace it with the one specified in **import**.

But importing may not be the best solution. All callers of the Chaosnet program probably want to refer to the same set of symbols: the symbols described in the documentation of the Chaosnet program. It is simplest if the Chaosnet program, rather than each caller, says which symbols they are.

Restricted inheritance allows the **chaos** package to specify which of its symbols should be inheritable. Then **file-access** can use package **chaos** and the desired symbols are available in it.

The inheritable symbols of a package such as **chaos** in this example are called *external*; the other symbols are *internal*. Symbols are internal by default. The function **export** is how symbols are made external. Only the external symbols of a package are inherited by other packages which use it. This is true of **global** as well; Only external symbols in **global** are inherited. Since **global** exists only for inheritance, every symbol in it is external; in fact, any symbol placed in **global** is automatically made external. **global** is said to be *autoexporting*. A few other packages with special uses, such as **keyword** and **fonts**, are autoexporting. Ordinary packages such as **chaos**, which programs are loaded in, should not be.

If a request is made to find a name in a package, first the symbols present directly in that package are searched. If the name is not found that way, then all the packages in the used-list are searched; but only external symbols are accepted. Internal symbols found in the used packages are ignored. If a new symbol needs to be created and put into the name space, it is placed directly in the specified package. New symbols are never put into the inherited packages.

The used packages of a package are not in any particular order. It does not make any difference which one is searched first, because they are *not allowed* to have any conflicts among them. If you attempt to set up an inheritance situation where a conflict would exist, you get an error immediately. You can then specify explicitly how to resolve the conflict. See section 27.7, page 647.

The packages used by the packages used are *not* searched. If package file-access uses package chaos and file mypackage uses package file-access, this does not cause mypackage to inherit anything from chaos. This is desirable: the Chaosnet functions for whose sake file-access uses chaos are not needed in the programs in mypackage simply to enable them to communicate with file-access. If it is desirable for mypackage to inherit from chaos, that can be requested explicitly.

These functions are used to set up and control package inheritance.

use-package *packages* &optional (*in-package* *package*)

Makes *in-package* inherit symbols from *packages*, which should be either a single package or name for a package, or a list of packages and/or names for packages.

This can cause a name conflict, if any of *packages* has a symbol whose name matches a symbol in *in-package*. In this case, an error is signaled, and you must resolve the conflict or abort.

unuse-package *packages* &optional (*in-package* *package*)

Makes *in-package* cease to inherit symbols from *packages*.

package-use-list *package*

Returns the list of packages used by *package*.

package-used-by-list *package*

Returns the list of packages which use *package*.

You can add or remove inheritance paths at any time, no matter what else you have done with the package.

These functions are used to make symbols external or internal in a package. By default, they operate on the current package.

export *symbols* &optional (*package* *package*)

Makes *symbols* external in *package*. *symbols* should be a symbol or string or a list of symbols and/or strings. The specified symbols or strings are interned in *package*, and the symbols found are marked external in *package*.

If one of the specified symbols is found by inheritance from a used package, it is made directly present in *package* and then marked external there. (We know it was already external in the package it was inherited from.)

Note that if a symbol is present directly in several packages, it can be marked external or internal in each package independently. Thus, it is the symbol's presence in a particular package which is external or not, rather than the symbol itself. export makes symbols external in whichever package you specify; if the same symbols are present directly in any other package, their status as external or internal in the other package is not affected.

unexport *symbols* &optional (*package* ***package***)
Makes *symbols* not be external in *package*. An error occurs if any of the symbols fails to be directly present in *package*.

package-external-symbols *package*
Returns a list of all the external symbols of *package*.

globalize *name-or-symbol* &optional (*into-package* **"GLOBAL"**)

Sometimes it will be discovered that a symbol which ought to be in **global** is not there, and the file defining it has already been loaded, thus mistakenly creating a symbol with that name in some other package. Creating a symbol in **global** would not fix the problem, since pointers to the misbegotten symbol already exist. Even worse, similarly named symbols may have been created mistakenly in other packages by code attempting to refer to the global symbol, and those symbols also are already pointed to. **globalize** is designed for use in correcting such a situation.

globalize *symbol-or-string* &optional (*package* **"GLOBAL"**)
If *name-or-symbol* is a name (a string), interns the name in *into-package* and then forwards together all symbols with the same name in all the packages that use *into-package* as well as in *into-package* itself. These symbols are forwarded together so that they become effectively one symbol as far as the value, function definition and properties are concerned. The value of the composite is taken from whichever of the symbols had a value; a proceedable error is signaled if multiple, distinct values were found. The function definition is treated similarly, and so is each property that any of the symbols has.

If *name-or-symbol* is a symbol, **globalize** interns that symbol in *into-package* and then forwards the other symbols to that one.

The symbol which ultimately is present in *into-package* is also exported.

27.6 Packages and Interning

The most important service of the package system is to look up a name in a package and return the symbol which has that name in the package's name space. This is done by the function **intern**, and is called *interning*. When you type a symbol as input, **read** converts your characters to the actual symbol by calling **intern**.

The function **intern** allows you to specify a package as the second argument. It can be specified by giving either the package object itself or a string or symbol that is a name for the package. **intern** returns three values. The first is the interned symbol. The second is a keyword that says how the symbol was found. The third is the package in which the symbol was actually found. This can be either the specified package or one of its used packages.

When you don't specify the second argument to **intern**, the current package, which is the value of the symbol ***package***, is used. This happens, in particular, when you call **read** and **read** calls **intern**. To specify the package for such functions to use, bind the symbol ***package*** temporarily to the desired package with **pkg-bind**.

There are actually four forms of the **intern** function: regular **intern**, **intern-soft**, **intern-local**, and **intern-local-soft**. **-soft** means that the symbol should not be added to the package if there isn't already one; in that case, all three values are **nil**. **-local** turns off inheritance; it means that the used packages should not be searched. Thus, **intern-local** can be used to cause shadowing. **intern-local-soft** is right when you want complete control over what packages to search and when to add symbols. All four forms of **intern** return the same three values, except that the **soft** forms return **nil nil nil** when the symbol isn't found.

intern *string-or-symbol* &optional (*pkg* ***package***)

The simplest case of **intern** is where *string-or-symbol* is a string. (It makes a big difference which one you use.) **intern** searches *pkg* and its used packages sequentially, looking for a symbol whose print-name is equal to *string-or-symbol*. If one is found, it is returned. Otherwise, a new symbol with *string-or-symbol* as print name is created, placed in package *pkg*, and returned.

The first value of **intern** is always the symbol found or created. The second value tells whether an existing symbol was found, and how. It is one of these four values:

:internal	A symbol was found present directly in *pkg*, and it was internal in *pkg*.
:external	A symbol was found present directly in *pkg*, and it was external in *pkg*.
:inherited	A symbol was found by inheritance from a package used by *pkg*. You can deduce that the symbol is external in that package.
nil	A new symbol was created

The third value returned by **intern** says which package the symbol found or created is present directly in. This is different from *pkg* if and only if if the second value is **:inherited.**

If *string-or-symbol* is a symbol, the search goes on just the same, using the print-name of *string-or-symbol* as the string to search for. But if no existing symbol is found, *string-or-symbol* itself is placed directly into *pkg*, just as **import** would do. No new symbol is created; *string-or-symbol itself* is the "new" symbol. This is done even if *string-or-symbol* is already present in another package. You can create arbitrary arrangements of sharing of symbols between packages this way.

Note: **intern** is sensitive to case; that is, it will consider two character strings different even if the only difference is one of upper-case versus lower-case. The reason that symbols get converted to upper-case when you type them in is that the reader converts the case of characters in symbols; the characters are converted to upper-case before **intern** is ever called. So if you call **intern** with a lower-case **"foo"** and then with an upper-case **"FOO"**, you won't get the same symbol.

intern-local *string-or-symbol* &optional (*pkg* ***package***)

Like **intern** but ignores inheritance. If a symbol whose name matches *string-or-symbol* is present directly in *pkg*, it is returned; otherwise *string-or-symbol* (if it is a symbol) or a new symbol (if *string-or-symbol* is a string) is placed directly in *pkg*.

intern-local returns second and third values with the same meaning as those of **intern**. However, the second value can never be **:inherited**, and the third value is always *pkg*.

The function **import** is implemented by passing the symbol to be imported to **intern-local**.

intern-soft *string* &optional (*pkg* ***package***)
find-symbol *string* &optional (*pkg* ***package***)

Like **intern** but never creates a symbol or modifies *pkg*. If no existing symbol is found, **nil** is returned for all three values. It makes no important difference if you pass a symbol instead of a string.

intern-soft returns second and third values with the same meaning as those of **intern**. However, if the second value is **nil**, it does not mean that a symbol was created, only that none was found. In this case, the third value is **nil** rather than a package.

find-symbol is the Common Lisp name for this function. The two names are synonymous.

intern-local-soft *string* &optional (*pkg* ***package***)

Like **intern-soft** but without inheritance. If a matching symbol is found directly present in *pkg*, it is returned; otherwise, the value is **nil**.

intern-local-soft returns second and third values with the same meaning as those of **intern**. However, if the second value is **nil**, it does not mean that a symbol was created, only that none was found. Also, it can never be **:inherited**. The third value is rather useless as it is either *pkg*, or **nil** if the second value is **nil**.

remob *symbol* &optional (*package* (**symbol-package** *symbol*))
unintern *symbol* &optional (*package* ***package***)

Both remove *symbol* from *package*. *symbol* itself is unaffected, but **intern** will no longer find it in *package*. *symbol* is not removed from any other package, even packages used by *package*, if it should be present in them. If *symbol* was present in *package* (and therefore, was removed) then the value is **t**; otherwise, the value is **nil**.

In **remob**, *package* defaults to the contents of the symbol's package cell, the package it belongs to. In **unintern**, *package* defaults to the current package. **unintern** is the Common Lisp version and **remob** is the traditional version.

If *package* is the package that *symbol* belongs to, then *symbol* is marked as uninterned: **nil** is stored in its package cell.

If a shadowing symbol is removed, a previously-hidden name conflict between distinct symbols with the same name in two used packages can suddenly be exposed, like a discovered check in chess. If this happens, an error is signaled.

27.7 Shadowing and Name Conflicts

In a package that uses **global**, it may be desirable to avoid inheriting a few standard Lisp symbols. Perhaps the user has defined a function **copy-list**, knowing that this symbol was not in **global**, and then a system function **copy-list** was created as part of supporting Common Lisp. Rather than changing the name in his program, he can *shadow* **copy-list** in the program's package. Shadowing a symbol in a package means putting a symbol in that package which hides any symbols with the same name which could otherwise have been inherited there. The symbol is explicitly marked as a *shadowing symbol* so that the name conflict does not result in an error.

Shadowing of symbols and shadowing of bindings are quite distinct. The same word is used for them because they are both examples of the general abstract concept of shadowing, which is meaningful whenever there is inheritance.

Shadowing can be done in the definition of a package (see page 652) or by calling the function **shadow**. **(shadow "COPY-LIST")** creates a new symbol named **copy-list** in the current package, regardless of any symbols with that name already available through inheritance. Once the new symbol is present directly in the package and marked as a shadowing symbol, the potentially inherited symbols are irrelevant.

shadow *names* &optional (*package* ***package***)

Makes sure that shadowing symbols with the specified names exist in *package*. *names* is either a string or symbol or a list of such. If symbols are used, only their names matter; they are equivalent to strings. Each name specified is handled independently as follows:

If there is a symbol of that name present directly in *package*, it is marked as a shadowing symbol, to avoid any complaints about name conflicts.

Otherwise, a new symbol of that name is created and interned in *package*, and marked as a shadowing symbol.

Shadowing must be done before programs are loaded into the package, since if the programs are loaded without shadowing first they will contain pointers to the undesired inherited symbol. Merely shadowing the symbol at this point does not alter those pointers; only reloading the program and rebuilding its data structures from scratch can do that.

If it is necessary to refer to a shadowed symbol, it can be done using a package prefix, as in **global:copy-list**.

Shadowing is not only for symbols inherited from **global**; it can be used to reject inheritance of any symbol. Shadowing is the primary means of resolving *name conflicts* in which there multiple symbols with the same name are available, due to inheritance, in one package.

Name conflicts are not permitted to exist unless a resolution for the conflict has been stated in advance by specifying explicitly which symbol is actually to be seen in package. If no resolution has been specified, any command which would create a name conflict signals an error instead.

For example, a name conflict can be created by **use-package** if it adds a new used package with its own symbol **foo** to a package which already has or inherits a different symbol with the same name **foo**. **export** can cause a name conflict if the symbol becoming external is now supposed to be inherited by another package which already has a conflicting symbol. On either occasion, if shadowing has not already been performed to control the outcome, an error is signaled and the useage or exportation does not occur.

The conflict is resolved—in advance, always—by placing the preferred choice of symbol in the package directly, and marking it as a shadowing symbol. This can be done with the function shadowing-import. (Actually, you can proceed from the error and specify a resolution, but this works by shadowing and retrying. From the point of view of the retried operation, the resolution has been done in advance.)

shadowing-import *symbols* &optional (*package* *package*)

Interns the specified symbols in *package* and marks them as shadowing symbols. *symbols* must be a list of symbols or a single symbol; strings are not allowed.

Each symbol specified is placed directly into *package*, after first removing any symbol with the same name already interned in *package*. This is rather drastic, so it is best to use shadowing-import right after creating a package, when it is still empty.

shadowing-import is primarily useful for choosing one of several conflicting external symbols present in packages to be used.

Once a package has a shadowing symbol named foo in it, any other potentially conflicting external symbols with name foo can come and go in the inherited packages with no effect. It is therefore possible to perform the use-package of another package containing another foo, or to export the foo in one of the used packages, without getting an error.

In fact, shadow also marks the symbol it creates as a shadowing symbol. If it did not do so, it would be creating a name conflict and would always get an error.

package-shadowing-symbols *package*

Returns the list of shadowing symbols of *package*. Each of these is a symbol present directly in *package*. When a symbol is present directly in more than one package, it can be a shadowing symbol in one and not in another.

27.8 Styles of Using Packages

The unsophisticated user need never be aware of the existence of packages when writing his programs. His files are loaded into package user by default, and keyboard input is also read in user by default. Since all the functions that unsophisticated users are likely to need are provided in the global package, which user inherits from, they are all available without special effort. In this manual, functions that are not in the global package are documented with colons in their names, and they are all external, so typing the name the way it is documented does work in both traditional and Common Lisp syntax.

However, if you are writing a generally useful tool, you should put it in some package other than user, so that its internal functions will not conflict with names other users use. If your program contains more than a few files, it probably should have its own package just on the chance that someone else will use it someday along with other programs.

If your program is large, you can use multiple packages to help keep its modules independent. Use one package for each module, and export from it those of the module's symbols which are reasonable for other modules to refer to. Each package can use the packages of other modules that it refers to frequently.

27.9 Package Naming

A package has one name, also called the *primary name* for extra clarity, and can have in addition any number of *nicknames*. All of these names are defined globally, and all must be unique. An attempt to define a package with a name or nickname that is already in use is an error.

Either the name of a package or one of its nicknames counts as a *name for* the package. All of the functions described below that accept a package as an argument also accept a name for a package (either as a string, or as a symbol whose print-name is the name). Arguments that are lists of packages may also contain names among the elements.

When the package object is printed, its primary name is used. The name is also used by default when printing package prefixes of symbols. However, when you create the package you can specify that one of the nicknames should be used instead for this purpose. The name to be used for this is called the *prefix name*.

Case is significant in package name lookup. Usually package names should be all upper case. read converts package prefixes to upper case except for quoted characters, just as it does to symbol names, so the package prefix will match the package name no matter what case you type it in, as long as the actual name is upper case: TV:FOO and tv:foo refer to the same symbol. |tv|:foo is different from them, and normally erroneous since there is no package initially whose name is 'tv' in lower case.

In the functions find-package and pkg-find-package, and others which accept package names in place of packages, if you specify the name as a string you must give it in the correct case:

```
(find-package "TV") => the tv package
(find-package "tv") => nil
```

You can alternatively specify the name as a symbol; then the symbol's pname is used. Since read converts the symbol's name to upper case, you can type the symbol in either upper or lower case:

```
(find-package 'TV) => the tv package
(find-package 'tv) => the tv package
```

since both use the symbol whose pname is "TV".

Relevant functions:

package-name *package*
Returns the name of *package* (as a string).

package-nicknames *package*

Returns the list of nicknames (strings) of *package*. This does not include the name itself.

package-prefix-print-name *package*

Returns the name to be used for printing package prefixes that refer to *package*.

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

Makes *new-name* be the name for *package*, and makes *new-nicknames* (a list of strings, possibly nil) be its nicknames. An error is signaled if the new name or any of the new nicknames is already in use for some other package.

find-package *name* &optional *use-local-names-package*

Returns the package which *name* is a name for, or nil if there is none. If *use-local-names-package* is non-nil, the local nicknames of that package are checked first. Otherwise only actual names and nicknames are accepted. *use-local-names-package* should be supplied only when interpreting package prefixes.

If *name* is a package, it is simply returned.

If a list is supplied as *name*, it is interpreted as a specification of a package name and how to create it. The list should look like

(*name super-or-use size*)

or

(*name options*)

If *name* names a package, it is returned. Otherwise a package is created by passing *name* and the *options* to make-package.

pkg-find-package *name* &optional *create-p use-local-names-package*

Invokes find-package on *name* and returns the package that finds, if any. Otherwise, a package may be created, depending on *create-p* and possibly on how the user answers. These values of *create-p* are meaningful:

nil	An error is signaled if an existing package is not found.
t	A package is created, and returned.
:find	nil is returned.
:ask	The user is asked whether to create a package. If he answers Yes, a package is created and returned. If he answers No, nil is returned.

If a package is created, it is done by calling make-package with *name* as the only argument.

This function is not quite for historical compatibility only, since certain values of *create-p* provide useful features.

sys:package-not-found (error) *Condition*

is signaled by pkg-find-package with second argument :error, nil or omitted, when the package does not exist.

The condition instance supports the operations :name and :relative-to: these return whatever was passed as the first and third arguments to pkg-find-package (the package name, and the package whose local nicknames should be searched).

The proceed types that may be available include

:retry	says to search again for the specified name in case it has become defined; if it is still undefined, the error occurs again.
:create-package	says to search again for the specified name, and create a package with that name (and default characteristics) if none exists yet.
:new-name	is accompanied by a name (a string) as an argument. That name is used instead, ignoring any local nicknames. If that name too is not found, another error occurs.
:no-action	(available on errors from within read) says to continue with the entire read as well as is possible without having a valid package.

27.9.1 Local Nicknames for Packages

Suppose you wish to test new versions of the Chaosnet and file access programs. You could create new packages test-chaos and test-file-access, and use them for loading the new versions of the programs. Then the old, installed versions would not be affected; you could still use them to edit and save the files of the new versions. But one problem must be solved: when the new file access program says "chaos:connect" it must get test-chaos:connect rather than the actual chaos:connect.

This is accomplished by making "CHAOS" a local nickname for "TEST-CHAOS" in the context of the package test-file-access. This means that the when a chaos: prefix is encountered while reading in package test-file-access, it refers to test-chaos rather than chaos.

Local nicknames are allowed to conflict with global names and nicknames; in fact, they are rarely useful unless they conflict. The local nickname takes precedence over the global name.

It is necessary to have a way to override local nicknames. If you (pkg-goto 'test-file-access), you may wish to call a function in chaos (to make use of the old, working Chaosnet program). This can be done using #: as the package prefix instead of just :. #: inhibits the use of local nicknames when it is processed. It always refers to the package which is globally the owner of the name that is specified.

#: prefixes are printed whenever the package name printed is also a local nickname in the current package; that is, whenever an ordinary colon prefix would be misunderstood when read back

These are the functions which manage local nicknames.

pkg-add-relative-name *in-pkg name for-pkg*
Defines *name* as a local nickname in *in-pkg* for *for-pkg*. *in-pkg* and *for-pkg* may be packages, symbols or strings.

pkg-delete-relative-name *in-pkg name*
Eliminates *name* as a local nickname in *in-pkg*.

Looking up local nicknames is done with find-package, by providing a non-nil *use-local-names-package* argument.

27.10 Defining Packages

Before any package can be referred to or made current, it must be defined. This is done with the special form defpackage, which tells the package system all sorts of things, including the name of the package, what packages it should use, its estimated size, and some of the symbols which belong in it. The defpackage form is recognized by Zmacs as a definition of the package name.

defpackage *name* &key ... *Macro*
Defines a package named *name*. The alternating keywords and values are passed, unevaluated, to *make-package* to specify the rest of the information about how to construct the package.

If a package named *name* already exists, it is modified insofar as this is possible to correspond to the new definition.

Here are the possible options and their meanings

nicknames	A list of nicknames for the new package. The nicknames should be specified as strings.
size	A number; the new package is initially made large enough to hold at least this many symbols before a rehash is needed.
use	A list of packages or names for packages which the new package should inherit from, or a single name or package. It defaults to just the **global** package.
prefix-name	Specifies the name to use for printing package prefixes that refer to this package. It must be equal to either the package name or one of the nicknames. The default is to use the name.
invisible	If non-nil, means that this package should not be put on the list ***all-packages***. As a result, find-package will not find this package, not by its name and not by any of its nicknames. You can make normal use of the package in all other respects (passing it as the second argument to intern, passing it to use-package to make other packages inherit from it or it from others, and so on).

export
import
shadow
shadowing-import

If any of these arguments is non-nil, it is passed to the function of the same name, to operate on the package. Thus, if *shadow* is ("FOO" "BAR"), then

```
(shadow this-package '("FOO" "BAR"))
```

is done.

You could accomplish as much by calling export, import, shadow or shadowing-import yourself, but it is clearer to specify all such things in one central place, the defpackage.

import-from If non-nil, is a list containing a package (or package name) followed by names of symbols to import from that package. Specifying *import-from* as (chaos "CONNECT" "LISTEN") is nearly the same as specifying *import* as (chaos:connect chaos:listen), the difference being that with *import-from* the symbols connect and listen are not looked up in the chaos package until it is time to import them.

super If non-nil, should be a package or name to be the superpackage of the new package. This means that the new package should inherit from that package, and also from all the packages that package inherits from. In addition, the superpackage is marked as autoexporting. Superpackages are obsolete and are implemented for compatibility only.

relative-names An alist specifying the local nicknames to have in this package for other packages. Each element looks like (*localname package*), where *package* is a package or a name for one, and *localname* is the desired local nickname.

relative-names-for-me

An alist specifying local nicknames by which this package can be referred to from other packages. Each element looks like (*package localname*), where *package* is a package name and *localname* is the name to refer to this package by from *package*.

For example, the system package eh could have been defined this way:

```
(defpackage "EH" :size 1200
  :use ("GLOBAL" "SYS") :nicknames ("DBG" "DEBUGGER")
  :shadow ("ARG"))
```

It has room initially for at least 1200. symbols, nicknames dbg and debugger, uses system as well as global, and contains a symbol named arg which is not the same as the arg in global. You may note that the function eh:arg is documented in this manual (see page 734), as is the function arg (see page 238).

The packages of our inheritance example (page 642) might have been defined by

```
(defpackage 'chaos :size 1000 :use '(sys global)
   :export ("CONNECT" "OPEN-STREAM" "LISTEN" ...
            "OPEN-STATE" "RFC-RECEIVED-STATE" ...))

(defpackage 'file-access :size 1500
   :use '(chaos global)
   :export ("OPEN-FILE" "CLOSE-FILE" "DELETE-FILE" ...)
   :import (chaos:connect chaos:open-state))

(defpackage 'mypackage :size 400
   :use '(file-access global))
```

It is usually best to put the package definition in a separate file, which should be loaded into the user package. (It cannot be loaded into the package it is defining, and no other package has any reason to be preferred.) Often the files to be loaded into the package belong to one or a few systems; then it is often convenient to put the system definitions in the same file (see chapter 28, page 660).

A package can also be defined by the package attribute in a file's `-*-` line. Normally this specifies which (existing) package to load, compile or edit the file in. But if the attribute value is a list, as in

```
-*-Package: (foo :size 300 :use (global system)); ...-*-
```

then loading, compiling or editing the file automatically creates package foo, if necessary with the specified options (just like defpackage options). No defpackage is needed. It is wise to use this feature only when the package is used for just a single file. For programs containing multiple files, it is good to make a system for them, and then convenient to put a defpackage near the defsystem.

make-package *name* &key *nicknames size use prefix-name invisible export shadow import shadowing-import import-from super relative-names relative-names-for-me*

Creates and returns new package with name *name*.

The meanings of the keyword arguments are described under defpackage (page 652).

pkg-create-package *name* &optional (*super* *package*) (*size* #o 200)

Creates a new package named *name* of size *size* with superpackage *super*. This function is obsolete.

kill-package *name-or-package*

Kills the package specified or named. It is removed from the list which is searched when package names are looked up.

package-declare *Macro*

package-declare is an older way of defining a package, obsolete but still used.

`(package-declare` *name superpackage size* `nil`
option-1 option-2 `...)`

creates a package named *name* with initial size *size*.

super specifies the *superpackage* to use for this package. Superpackages were an old way of specifying inheritance; it was transitive, all symbols were inherited, and only one inheritance path could exist. If *super* is global, nothing special needs to be done; otherwise, the old superpackage facility is simulated using the *super* argument to make-package.

body is now allowed to contain only these types of elements:

(shadow *names*)
: Passes the names to the function SHADOW.

(intern *names*)
: Converts each name to a string and interns it in the package.

(refname *refname packagename*)
: Makes *refname* a local nickname in this package for the package named *packagename*.

(myrefname *packagename refname*)
: Makes *refname* a local nickname in the package named *packagename* for this package. If *packagename* is "GLOBAL", makes *refname* a global nickname for this package.

(external *names*)
: Does nothing. This controlled an old feature that no longer exists.

27.11 Operating on All the Symbols in a Package

To find and operate on every symbol present or available in a package, you can choose between iteration macros that resemble dolist and mapping functionals that resemble mapcar.

Note that all constructs that include inherited symbols in the iteration can process a symbol more than once. This is because a symbol can be directly present in more than one package. If it is directly present in the specified package and in one or more of the used packages, the symbol is processed once each time it is encountered. It is also possible for the iteration to include a symbol that is not actually available in the specified package. If that package shadows symbols present in the packages it uses, the shadowed symbols are processed anyway. If this is a problem, you can explicitly use intern-soft to see if the symbol handed to you is really available in the package. This test is not done by default because it is slow and rarely needed.

do-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-local-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-external-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each external symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-local-external-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each external symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-all-symbols (*var result-form*) *body...* *Macro*

Executes *body* once for each symbol present in any package. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once.

mapatoms *function* &optional (*package* *package*) (*inherited-p* t)

function should be a function of one argument. mapatoms applies *function* to all of the symbols in *package*. If *inherited-p* is non-nil, then the function is applied to all symbols available in *package*, including inherited symbols.

mapatoms-all *function* &optional (*package* "GLOBAL")

function should be a function of one argument. mapatoms-all applies *function* to all of the symbols in *package* and all other packages which use *package*.

It is used by such functions as apropos and who-calls (see page 791)
Example:

```
(mapatoms-all
  #'(lambda (x)
      (and (alphalessp 'z x)
           (print x))))
```

27.12 Packages as Lisp Objects

A package is a conceptual name space; it is also a Lisp object which serves to record the contents of that name space, and is passed to functions such as intern to identify a name space.

packagep *object*

t if object is a package.

all-packages *Variable*

The value is a list of all packages, except for invisible ones (see the *invisble* argument to make-package, page 654).

`list-all-packages`
A Common Lisp function which returns `*all-packages*`.

`pkg-global-package` *Constant*
`pkg-system-package` *Constant*
`pkg-keyword-package` *Constant*
Respectively, the packages named global, system and keyword.

`describe-package` *package*
Prints everything there is to know about *package*, except for all the symbols interned in it. *package* can be specified as a package or as the name of one.

To see all the symbols interned in a package, do

```
(mapatoms 'print package)
```

27.13 Common Lisp and Packages

Common Lisp does not have defpackage or `-*-` lines in files. One is supposed to use the function `in-package` to specify which package a file is loaded in.

`in-package` *name* &key *nicknames use*
Creates a package named *name*, with specified nicknames and used packages, or modifies an existing package named *name* to have those nicknames and used packages.

Then `*package*` is set to this package.

Writing a call to `in-package` at the beginning of the file causes `*package*` to be set to that package for the rest of the file.

If you wish to use this technique for the sake of portability, it is best to have a `-*-` line with a package attribute also. While `in-package` does work for loading and compilation of the file, Zmacs does not respond to it.

In Common Lisp, the first argument to intern or find-symbol is required to be a symbol.

27.14 Initialization of the Package System

This section describes how the package system is initialized when generating a new software release of the Lisp Machine system; none of this should affect users.

The cold load, which contains the irreduceable minimum of the Lisp system needed for loading the rest, contains the code for packages, but no packages. Before it begins to read from the keyboard, it creates all the standard packages based on information in si:initial-packages, applying make-package to each element of it. At first all of the packages are empty. The symbols which belong in the packages global and system are recorded on lists which are made from the files SYS: SYS2; GLOBAL LISP and SYS: SYS2; SYSTEM LISP. Symbols referred to in the cold load which belong in packages other than si have strings (package names) in their package slots; scanning through the area which contains all the symbols, the package initializer puts each such symbol into the package it specifies, and all the rest into **si** unless they are already in **global** or **system**.

27.15 Initial Packages

The initially present packages include:

global	Contains advertised global functions.
user	The default current package for the user's type-in.
sys or **system**	Contains internal global symbols used by various system programs. Many system packages use **system**.
si or **system-internals**	Contains subroutines of many advertised system functions. Many files of the Lisp system are loaded in **si**.
compiler	Contains the compiler. **compiler** uses **sys**.
fs or **file-system**	Contains the code that deals with pathnames and accessing files. **fs** uses **sys**.
eh or dbg	Contains the error handler and the debugger. Uses **sys**.
cc or cadr	Contains the program that is used for debugging another machine. Uses **sys**.
chaos	Contains the Chaosnet controller. Uses **sys**.
tv	Contains the window system. Uses **sys**.
zwei	Contains the editor.
format	Contains the function **format** and its associated subfunctions.
cli	(Common Lisp Incompatible) contains symbols such as **cli:member** which the same pname as symbols in **global** but incompatible definitions.

There are quite a few others, but it would be pointless to list them all.

Packages that are used for special sorts of data:

fonts	Contains the names of all fonts.
format	Contains the keywords for **format**, as well as the code.
keyword	Contains all keyword symbols, symbols always written with a plain colon as a prefix. These symbols are peculiar in that they are automatically given themselves as values.

Here is a picture depicting the initial package inheritance structure

```
                       global                      keyword
                         |
  /-------------------------------------\          fonts
  |      |          |          |        |
user  zwei       system      format  (etc)         cli
                    |
            /-----------------------------------\
            |          |     |     |     |      |
    system-internals   eh  chaos  cadr  fs   compiler
```