24. 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 the user uses during his 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 the user 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 declare 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's name space and would refer to the Arpanet program's **allocate-pbuf**.

To understand what is going on here, you should keep in mind how Lisp reading and loading works. When a file is gotten into the Lisp Machine, either by being read or by being fasloaded, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it calls **intern** to look up that string in some name space and find a corresponding symbol. The package system arranges that the correct name space is used whenever a file is loaded.

24.1 The Organization of Name Spaces

We could simply let every name space be implemented as one obarray, e.g. one big table of symbols. The problem with this is that just about every name space wants to include the whole Lisp language: **car**, **cdr**, and so on should be available to every program. We would like to share the main Lisp system between several name spaces without making many copies.

Instead of making each name space be one big array, we arrange packages in a tree. Each package has a "superpackage" or "parent", from which it "inherits" symbols. Also, each package has a table, or "obarray", of its own additional symbols. The symbols belonging to a package are simply those in the package's own obarray, followed by those belonging to the superpackage. The root of the tree of packages is the package called **global**, which has no superpackage. **global** contains **car** and **cdr** and all the rest of the standard Lisp system. In our example, we might have two other packages called **chaos** and **arpa**, each of which would have **global** as its parent. Here is a picture of the resulting tree structure:

```
                    global
                      |
       /----------------------------\
       |                            |
     chaos                         arpa
```

In order to make the sharing of the **global** package work, the **intern** function is made more complicated than in basic Lisp. In addition to the string or symbol to intern, it must be told which package to do it in. First it searches for a symbol with the specified name in the obarray of the specified package. If nothing is found there, **intern** looks at its superpackage, and then at the superpackage's superpackage, and so on, until the name is found or a root package such as **global** is reached. When **intern** reaches the root package, and doesn't find the symbol there either, it decides that there is no symbol known with that name, and adds a symbol to the originally specified package.

Since you don't normally want to worry about specifying packages, **intern** normally uses the "current" package, which is the value of the symbol **package**. This symbol serves the purpose of the symbol **obarray** in Maclisp.

Here's how that 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 on the **chaos** package. If there is a reference to some well known global symbol such as **append**, **intern** would look for "**append**" on the **chaos** package, not find it, look for "**append**" on **global**, and find the regular Lisp **append** symbol, and return that. If, however, there is a reference to a symbol that the user made up himself (say it is called **get-packet**), the first time he uses it, **intern** won't find it on either **chaos** or **global**. So **intern** will make a new symbol named get-packet, and install it on the chaos package. When get-packet is referred to later in the Chaosnet program, intern will find get-packet on the chaos package.

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 will *not* get the same one the Chaosnet program got, because the chaos package is not being searched. Rather, the arpa and global packages are getting searched. So intern will create a new get-packet and install it on the arpa package.

So what has happened is that there are two get-packets: one for chaos and one for arpa. The two programs are loaded together without name conflicts.

24.2 Shared Programs

Now, a very important feature of the Lisp Machine is that of "shared programs"; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with PDP-10 system programs, in which Roman numerals have been independently reimplemented several times (and the ITS filename parser several dozen times).

For example, the routines to manipulate a robot arm might be a separate program, residing in a package named arm. If we have a second program called blocks (the blocks world, of course) which wanted to manipulate the arm, it would want to call functions which are defined on the arm obarray, and therefore not in blocks's own name space. Without special provision, there would be no way for any symbols not in the blocks name space to be part of any blocks functions.

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 will read in the next Lisp object with package bound to that package. The way blocks would call a function named go-up defined in arm would be by asking to call arm:go-up, because go-up would be interned on the arm package. What arm:go-up means precisely is "the symbol named go-up in the name space of the package arm."

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

An important question that should occur at this point is how the names of packages are associated with their obarrays and other data. This is done by means of the "refname-alist" that each package has. This alist associates strings called *reference names* or *refnames* with the packages they name. Normally, a package's refname-alist contains an entry for each subpackage, associating the subpackage with its name. In addition, every package has its own name defined as a refname, referring to itself. However, the user can add any other refnames, associating them with any packages he likes. This is useful when multiple versions of a program are loaded into different packages. Of course, each package inherits its superpackage's refnames just as it does symbols.

In our example, since arm is a subpackage of global, the name arm is on global's refname-alist, associated with the arm package. Since blocks is also a subpackage of global, when arm:go-up is seen the string "arm" is found on global's refname alist.

When you want to refer to a symbol in a package which you and your superpackages have no refnames for—say, a subpackage named foo of a package named bar that is under global—you can use multiple colons. For example, the symbol finish in that package foo could be referred to as foo:bar:finish. What happens here is that the second name, bar, is interpreted as a refname in the context of the package foo. Alternatively, you can give the foo:bar package a refname directly under global, even though global is not its superior as a package. This is usually done by using myrefname in the package declaration (page 517).

24.3 Declaring Packages

Before any package can be referred to or loaded, it must be declared. This is done with the special form package-declare, which tells the package system all sorts of things, including the name of the package, the place in the package hierarchy for the new package to go, its estimated size, and some of the symbols which belong in it.

Here is a sample declaration:

```
(package-declare foo global 1000
         ()
         (shadow array-push adjust-array-size))
```

What this declaration says is that a package named foo should be created as an inferior of global, the package which contains advertised global symbols. Its obarray should initially be large enough to hold 1000 symbols, though it will grow automatically if that isn't enough. Unless there is a specific reason to do otherwise, you should make all of your packages direct inferiors of global. The size you give is increased slightly to be a good value for the hashing algorithm used.

After the size comes the "file-alist", which is given as () in the example. This is an obsolete feature, which is not normally used. The "system"-defining facilities should be used instead. See chapter 25, page 520.

Finally, the foo package "shadows" array-push and adjust-array-size. What shadowing means is that the foo package should have its own versions of those symbols, rather than inheriting its superpackage's versions. Symbols by these names will be added to the foo package even though there are symbols on global already with those names. This allows the foo package to redefine those functions for itself without redefining them in the global package for everyone else.

Certain other things may be found in the declarations of various internal system packages. They are arcane and needed only to compensate for the fact that parts of those packages are actually loaded before the package system is. They should not be needed by any user package.

Your package declarations should go into separate files containing only package declarations. Group them however you like, one to a file or all in one file. Such files can be read with load. It doesn't matter what package you load them into, so use user, since that has to be safe.

If the declaration for a package is read in twice, no harm is done. If you edit the size to replace it with a larger one, the package will be expanded. At the moment, however, there is no way to change the list of shadowings; such changes will be ignored. Also, you can't change the superpackage. If you edit the superpackage name and read the declaration in again, you will create a new, distinct package without changing the old one.

package-declare *Macro*

The package-declare macro is used to declare a package to the package system. Its form is:

```
(package-declare name superpackage size
                 file-alist option-1 option-2 ...)
```

The interpretation of the declaration is complicated; see section 24.3, page 509.

describe-package *package-name*

(describe-package *package-name*) is equivalent to (describe (pkg-find-package *package-name*)); that is, it describes the package whose name is *package-name*.

24.4 Packages and Writing Code

The unsophisticated user need never be aware of the existence of packages when writing his programs. He should just load all of his programs into the package user, which is also what console type-in is interned in. Since all the functions that users are likely to need are provided in the global package, which is user's superpackage, they are all available. In this manual, functions that are not on the global package are documented with colons in their names, so typing the name the way it is documented will work.

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 you are loading your programs into packages other than user, there are special constructs that you will need to know about.

One time that you as the programmer must be aware of the existence of packages is when you want to use a function or variable in another package. To do this, write the name of the package, a colon, and then the name of the symbol, as in eine:ed-get-defaulted-file-name. You will notice that symbols in other packages print out that way, too. Sometimes you may need to refer to a symbol in a package whose superior is not global. When this happens, use multiple colons, as in foo:bar:ugh, to refer to the symbol ugh in the package named bar which is under the package named foo.

Another time that packages intrude is when you use a "keyword", for instance, when you check for eqness against a constant symbol, or pass a constant symbol to someone else who will check for it using eq. This includes using the symbol as either argument to get. In such cases, the usual convention is that the symbol should reside in the user package, rather than in the package with which its meaning is associated. To make it easy to specify user, a colon before a symbol, as in :select, is equivalent to specifying user by name, as in user:select. Since the user package has no subpackages, putting symbols into it will not cause name conflicts.

Why is this convention used? Well, consider the function make-array, which takes one required argument followed by any number of keyword arguments. For example,

```
(make-array 100 'leader-length 10 'type art-string)
```

specifies, after the first required argument, two options with names leader-length and type and values 10 and art-string. The file containing this function's definition is in the system-internals package, but the function is available to everyone without the use of a colon prefix because the symbol make-array is itself inherited from global. But all the keyword names, such as type, are short and should not have to exist in global. However, it would be a shame if all callers of make-array had to specify system-internals: before the name of each keyword. After all, those callers can include programs loaded into user, which should by rights not have to know about packages at all. Putting those keywords in the user package solves this problem. The correct way to type the above form would be

```
(make-array 100 ':leader-length 10 ':type art-string)
```

Exactly when should a symbol go in user? Most symbols known specially by system functions are in user. Symbols used as keywords for arguments by any function should usually be in user, to keep things consistent. However, when a program uses a specific property name to associate its own internal memoranda with symbols passed in from outside, the property name should belong to the program's package, so that two programs using the same property name in that way don't conflict.

24.5 Shadowing

Suppose the user doesn't like the system nth function; he might be a former Interlisp user and expect a completely different meaning from it. Were he to say (defun nth ---) in his program (call it snail) he would clobber the global symbol named "nth" and affect the "nth" in everyone else's name space. (Actually, since he would be redefining this function in a different file from the original definition, he would get a warning and a query.)

In order to allow the snail package to have its own (defun nth ---) without interfering with the rest of the Lisp environment, it must "shadow" out the global symbol "nth" by putting a new symbol named "nth" on its own obarray. Normally, this is done by writing (shadow nth) in the declaration of the snail package. Since intern looks on the subpackage's obarray before global, it will find the programmer's own nth and never the global one. Since the global one is now impossible to see, we say it has been "shadowed."

Having shadowed nth, we may sometimes need to refer to the global definition. This can be done by writing global:nth. This works because the refname global is defined in the global package as a name for the global package. Since global is the superpackage of the snail package, all refnames defined by global, including "global", are available in snail.

24.6 Packages and Interning

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 the name of the package. **intern** returns three values. The first is the interned symbol. The second is **t** if the symbol is old (was already present, not just added to the obarray). The third is the package in which the symbol was actually found. This can be either the specified package or one of its superiors.

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**. To specify the package for such functions to use, bind the symbol **package** temporarily to the desired package with **let**. The function **pkg-find-package** may be used to obtain the package with a given name. While most functions that use packages will do this themselves, it is better to do it only once when **package** is bound. The function **pkg-goto** sets **package** to a package specified by a string. This is unclean if done in a program, but is useful for "putting the keyboard inside" a package when you are debugging.

package *Variable*

The value of **package** is the current package. Many functions such as **read** always operate on this package, and other functions such as **intern** which accept a package as an optional argument default to this one.

pkg-goto &optional *pkg*

pkg may be a package or the name of a package. *pkg* is made the current package. It defaults to the **user** package.

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))
```

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** means that the superpackages 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* &optional (*pkg* **package**)

intern searches *pkg* and its superpackages sequentially, looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and the package on which the symbol is interned. If it does not find one, it creates a new symbol with a print name of *string*, interns it into the package *pkg*, and returns the new symbol, **nil**, and *pkg*.

If *string* is not a string but a symbol, intern searches for a symbol with the same print-name. If it doesn't find one, it interns *string*—rather than a newly-created symbol—in *pkg* (even if it is also interned in some other package) and returns it.

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 (unlike most string comparisons elsewhere in the Lisp Machine system). 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* &optional (*pkg* package)

intern searches *pkg* (but *not* its superpackages), looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, t, and *pkg* If it does not find one, it creates a new symbol with a print name of *string*, and returns the new symbol, nil, and *pkg*.

If *string* is not a string but a symbol, and no symbol with that print-name is already interned in *pkg*, intern-local interns *string*—rather than a newly-created symbol—in *pkg* (even if it is also interned in some other package) and returns it.

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

intern searches *pkg* and its superpackages sequentially, looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, t, and the package on which the symbol is interned. If it does not find one, it returns nil, nil, and nil.

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

intern searches *pkg* (but *not* its superpackages), looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, t, and *pkg* If it does not find one, it returns nil, nil, and nil.

Each symbol remembers which package it belongs to. While you can intern a symbol in any number of packages, the symbol will only remember one: normally, the first one it was interned in, unless you clobber it. This package is available as (symbol-package *symbol*). If the value is nil, the symbol believes that it is uninterned.

The printer also implicitly uses the value of package when printing symbols. If slashification is on, the printer tries to print something such that if it were given back to the reader, the same object would be produced. If a symbol that is not in the current name space were just printed as its print name and read back in, the reader would intern it on the wrong package, and return the wrong symbol. So the printer figures out the right colon prefix so that if the symbol's printed representation were read back in to the same package, it would be interned correctly. The prefix is only printed if slashification is on, i.e. prin1 prints it and princ does not.

remob *symbol* &optional *package*

remob removes *symbol* from *package* (the name means "REMove from OBarray"). *symbol* itself is unaffected, but intern will no longer find it on *package*. remob is always "local", in that it removes only from the specified package and not from any superpackages. It returns t if the symbol was found to be removed. *package* defaults to the contents of the symbol's package cell, the package it is actually in. (Sometimes a symbol can be in other packages also, but this is unusual.)

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.

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

function should be a function of one argument. mapatoms applies *function* to all of the symbols in *package*. If *superiors-p* is t, then the function is also applied to all symbols in *package*'s superpackages. Note that the function will be applied to shadowed symbols in the superpackages, even though they are not in *package*'s name space. If that is a problem, *function* can try applying intern in *package* on each symbol it gets, and ignore it if it is not eq to the result of intern; this measure is rarely needed.

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 of *package*'s subpackages. Since *package* defaults to the global package, this normally gets at all of the symbols in all packages. It is used by such functions as apropos and who-calls (see page 640)

Example:

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

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

pkg-create-package creates and returns a new package. Usually packages are created by package-declare, but sometimes it is useful to create a package just to use as a hash table for symbols, or for some other reason.

If *name* is a list, its first element is taken as the package name and the second as the program name; otherwise, *name* is taken as both. In either case, the package name and program name are coerced to strings. *super* is the superpackage for this package; it may be nil, which is useful if you want the package only as a hash table, and don't want it to interact with the rest of the package system. *size* is the size of the package; as in package-declare it is rounded up to a "good" size for the hashing algorithm used.

pkg-kill *pkg*

pkg may be either a package or the name of a package. The package should have a superpackage and no subpackages. pkg-kill takes the package off its superior's subpackage list and refname alist.

pkg-find-package *x* &optional (*create-p* nil) (*under* "global")

pkg-find-package tries to interpret *x* as a package. Most of the functions whose descriptions say "... may be either a package or the name of a package" call pkg-find-package to interpret their package argument.

If *x* is a package, pkg-find-package returns it. Otherwise it should be a symbol or string, which is taken to be the name of a package. The name is looked up on the refname alists of package and its superpackages, the same as if it had been typed as part of a colon prefix. If this finds the package, it is returned. Otherwise, *create-p* controls what happens. If *create-p* is nil, an error is signalled. If *create-p* is :find, nil is returned. If *create-p* is :ask the user is asked whether to create it. Otherwise, a new package is created, and installed as an inferior of *under*.

A package is implemented as a structure, created by defstruct. The following accessor defsubsts are available on the global package:

pkg-name	The name of the package, as a string.
pkg-refname-alist	The refname alist of the package, associating strings with packages.
pkg-super-package	The superpackage of the package.

24.7 Status Information

The current package—where your type-in is being interned—is always the value of the symbol package. A package is a named structure that prints out nicely, so examining the value of package is the best way to find out what the current package is. (It is also displayed in the who-line.) Normally, it should be user, except when inside compilation or loading of a file belonging to some other package.

To get more information on the current package or any other, use the function describe-package. Specify either a package object or a string that is a refname for the desired package as the argument. This will print out everything except a list of all the symbols in the package. If you want *that*, use (mapatoms 'print *package* nil). describe of a package will call describe-package.

24.8 Packages, Loading, and Compilation

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 "file attribute list" (see section 21.9.2, page 438). 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.

24.9 Subpackages

Usually, each independent program occupies one package, which is directly under **global** in the hierarchy. But large programs, such as Macsyma, are usually made up of a number of sub-programs, which are maintained by a small number of people. We would like each sub-program to have its own name space, since the program as a whole has too many names for anyone to remember. So we can make each sub-program into its own package. However, this practice requires special care.

It is likely that there will be a fair number of functions and symbols that should be shared by all of the sub-programs of Macsyma. These symbols should reside in a package named **macsyma**, which would be directly under **global**. Then each part of **macsyma** (for instance, **sin**, **risch**, **input**, and so on) would have its own package, with the **macsyma** package as its superpackage. To do this, first declare the **macsyma** package, and then declare the **risch**, **sin**, etc. packages, specifying **macsyma** as the superpackage for each of them. This way, each sub-program gets its own name space. All of these declarations would probably be together in a file called something like **MACPKG**.

However, to avoid a subtle pitfall, it is necessary that the **macsyma** package itself contain no files, only a set of symbols specified at declaration time. This list of symbols is specified using **shadow** in the declaration of the **macsyma** package. The symbols residing in the **macsyma** package can have values and definitions, but these must all be supplied by files in **macsyma's** subpackages. Note that this is exactly the same treatment that **global** receives: all its functions are actually defined in files that are loaded into **system-internals (si)**, **compiler**, etc.

To demonstrate the full power and convenience of this scheme, suppose there were a second huge program called **owl** that also had a subprogram called **input** (which, presumably, does all of the **inputting** for **owl**), and one called **database**. Then a picture of the hierarchy of packages would look like this:

```
                         global
                           |
                 /-------------------------------\
                 |                               |
              macsyma                           owl
                 |                               |
  -----------------------------          ----------------------
  | | |      |        |        |         |         |        | | |
(others)   risch     sin     input     input   database   (others)
```

Now, the risch program and the sin program both do integration, and so it would be natural for each to have a function called integrate. From inside sin, sin's integrate would be referred to as "integrate" (no prefix needed), while risch's would be referred to as "risch:integrate". Similarly, from inside risch, risch's own integrate would be called "integrate", whereas sin's would be referred to as "sin:integrate".

If sin's integrate were a recursive function, the implementor would be referring to it from within sin itself, and would be happy that he need not type out "sin:integrate" every time; he could just say "integrate".

From inside the macsyma package or any of its other sub-packages, the two functions would be referred to as "sin:integrate" and as "risch:integrate". From anywere else in the hierarchy, they would have to be called "macsyma:sin:integrate" and "macsyma:risch:integrate".

Similarly, assume that each of the input packages has a function called get-line. From inside macsyma or any of macsyma's subprograms (other than input), the relevant function would be called input:get-line and the irrelevant one owl:input:get-line. The converse is true for owl and its sub-programs. Note that there is no problem arising from the fact that both owl and macsyma have subprograms of the same name (input).

You might also want to put Macsyma's get-line function on the macsyma package. Then, from anywehere inside Macsyma, the function would be called get-line; from the owl package and subpackages it could be referred to as macsyma:get-line.

You can give a second-level subpackage a globally available name of its own by using myrefname in the package declaration. For example,

```
        (myrefname global macinput)
```

in the declaration of the macsyma:input package would give it the additional refname macinput that can be used from any package under global.

24.10 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.

When the world begins to be loaded, there is no package system. There is one "obarray", whose format is different from that used by the package system. When the package system is loaded and initialized, it is necessary to split the symbols of the old-style obarray up among the various initial packages.

The first packages created by initialization are the most important ones: global, system, user, and system-internals. All of the symbols already present are placed in one of those packages. By default, a symbol goes into system-internals. Only those placed on special lists go into one of the others. These lists are the file SYS: SYS2; GLOBAL LISP of symbols which belong in global, and the file SYS: SYS2; SYSTEM LISP of symbols which go in system.

After the four basic packages exist, the package system's definition of intern is installed, and packages exist. Then, the other system packages format, compiler, zwei, etc. are declared in almost the normal manner. The exception is that a few of the symbols present before packages exist really belong in one of these packages. Their package declarations contain calls to forward and borrow, which exist only for this purpose and are meaningful only in package declarations, and are used to move the symbols as appropriate. These declarations are kept in the file SYS: SYS; PKGDCL LISP.

globalize *symbol* &optional (*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 will not fix the problem, since the existing symbol will shadow it. Worse, functions in other packages which should have used the global symbol may have shadowed it in their own packages.

When this happens, you can correct the situation by doing (globalize "*symbol-name*"). This function creates a symbol with the desired name in global, merges whatever value, function definition, and properties can be found on symbols of that name together into the new symbol (complaining if there are conflicts), and forwards those slots of the existing symbols to the slots of the new one using one-q-forward pointers, so that they will appear to be one and the same symbol as far as value, function definition, and property list are concerned. They cannot all be made eq to each other, but globalize does the next-best thing: it takes an existing symbol from user, if there is one, to put it in global. Since people who check for eq are normally supposed to specify user anyway, they will not perceive any effect from moving the symbol from user into global.

If globalize is given a symbol instead of a string as argument, the exact symbol specified is put into global. You can use this when a symbol in another package, which should have been inherited from global, is being checked for with eq—as long as there are not *two* different packages doing so. Usually, the symbol is just a function name or a variable, and this problem does not arise.

If the argument *package* is specified, then the symbol is moved into that package from all its subpackages, rather than into **global**.

24.11 Initial Packages

The initially present packages include:

global	Contains advertised global functions.
user	Used for interning the user's type-in. Contains all keyword symbols.
sys or **system**	Contains internal global symbols used by various system programs. **global** is for symbols global to the Lisp language, while **system** is for symbols global to the Lisp Machine operating system.
si or **system-internals**	Contains subroutines of many advertised system functions. **si** is a subpackage of **sys**.
compiler	Contains the compiler. **compiler** is a subpackage of **sys**.
fs or **file-system**	Contains the code that deals with pathnames and accessing files.
eh or **dbg**	Contains the error handler and the debugger.
cc or **cadr**	Contains the program that is used for debugging another machine.
zwei	Contains the editor.
chaos	Contains the Chaosnet controller.
tv	Contains the window system.
format	Contains the function **format** and its associated subfunctions.

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.

Here is a picture depicting the initial package hierarchy:

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