Packages in Common Lisp, a tutorial

Francis Sergeraert

January 2014

1 Introduction.

It is not possible to seriously program without using *identifiers*, allowing the programmer to locate, process and modify various datas in a convenient way, with names more or less descriptive, pointing to the data, depending in some way on the context. The standard *modular* technology divides large programs in *modules*; the modules can be relatively independent of each other, but conversely, they must *communicate*. Most programming languages propose methods to treat such an organization. For example, the set of identifiers of a module could be divided in two parts, the *private* identifiers, known only from this module, and the *public* identifiers, on the contrary known everywhere.

In this domain as in many others, Common Lisp is by far the most powerful and flexible programming language. With the cost of a relative complexity, needing a good knowledge of the corresponding *definitions* in the language. These definitions are not so easy to understand if the Ansi definition of the language is the only source. An identifier in Lisp is a *symbol*, which has by itself a rich structure, in particular, which continues to be "alive" at running time. The modular structure of a Lisp program is defined through the *packaging* process, allowing to divide a large program in several files, each file knowing all the symbols of some package, the current *default package*, and only the *external* (public) symbols of some other packages. A symbol which is not external in a package is *internal* (private). Also, for some reasons, a package can also *import* (that is, "see") an arbitrary collection of symbols, internal or external, of the other packages. Even if usually it is not really appropriate, the whole structure of such an organization can be arbitrarily dynamically modified if necessary, following the general spirit of Lisp.

These notes are devoted to a *mathematical* definition of this process, which could help a Lisp programmer to use it more lucidly. A basic but *inexact* presentation of the packaging system is the following:

- A *package* is a Lisp object as a number, a list, a character string. In particular the *type* `package` is defined. A package can be the value of a symbol, given as argument to a function, returned by a function, etc.
- A *symbol* is a Lisp object by itself. In particular the *type* `symbol` is defined. Same comments as for the packages, in particular a symbol can be the value

of another symbol, or even of itself.
- The symbols are divided in packages.
- At some time of a Lisp session, some package is the *default package* and, first approximation, only the symbols of this package are *accessible.* The default package can be modified at any time by the user or by a program.
- However, if necessary, there exist different methods to reach the symbols of a package different from the default package. Conversely a package may make its symbols more or less easily accessible from the other packages.

This first approximation of the packaging process gives the general style of its organization, but the deep nature of this organization is not at all this one. It is essentially *false* that the symbols are divided in packages. Or more exactly, it is essentially true (!), but this division in packages, via the home package of each symbol, plays only a *minor* role, which can be totally forgotten by the programmer without any risk. This notion of home package is terribly misleading for the beginners when they have to design non-trivial packaging structures. In a Lisp environment, you have on one hand the space of symbols, and on the other hand the space of packages, and it is only *inside* the packages that something can be read which looks *a little* like a division of symbols in packages, but the symbols themselves essentially *ignore* this division. Our subject consists in explaining this organization.

The Ansi definition of the packaging process is a *complete* description of this organization; in a sense, nothing is to be added. But experience shows it quickly becomes difficult to understand it when the situation is a little complex, easily generating erroneous programming choices and mysterious bugs. The Ansi definition of packaging is complete, it is a perfect *Reference Guide.* In this domain as in many others, a reference guide is not enough, a *User Guide* might also be useful. In this respect, an elementary *mathematical* description of the Common Lisp packaging system could help programmers to understand it, and then to use it more easily and also more powerfully. In every domain, when the usual mathematical language can be used and understood, the situation becomes better.

In a first step, we reread the Ansi definition of the packaging process, adding in parallel a mathematical description of the main concepts. Finally we give a complete mathematical definition of this system as a set of mathematical formulas.

2 Main sets of objects.

The first component of the packaging system is made of three *disjoint* sets:

- The set St of *strings.*
- The set P of *packages.*
- The set Sm of *symbols.*

In our description of the packaging system, the status of these sets are different. Think the (infinite) set St of *strings* is made of *all* the character strings which *can* be used by an user or a program at any time; any arbitrary string can be used at any time, whatever is the history of the Lisp session we are working in. In the packaging process, most often, a particular string is used for a short time, mainly during the read-eval-print loop, in fact more specifically when the Lisp reader works. A string is just an intermediate process to define a connection between the user and the Lisp session, more precisely the current environment. On the contrary, in a given environment, at some time of a Lisp session, some finite set P of *specific* packages is installed in this environment, and also a finite set Sm of *specific* symbols, those packages and symbols which can *then* be used, all being the consequence of the history of the current Lisp session. Various Lisp functions allow the user to modify the sets P and Sm and the way they are combined to each other. In other words, consider the set of strings St is constant, vast and independent of the environment, while the sets P and Sm on the contrary are restricted, vary during a Lisp session and are an essential component of the *current* environment.

3 Packages.

There is a simple correspondence between strings and packages, almost bijective, convenient to start our subject. First, at any time of a Lisp session, a *default package* is defined, located by the global symbol `*package*`:

```
> *package* ✠
#<The COMMON-LISP-USER package>
```

You read: the *value* of the *symbol* `*package*` is the *package* whose *name* is the string `"COMMON-LISP-USER"`. The types *symbol* and *package* are defined, defining two disjoint sets of machine objects. A symbol is most often used as a convenient intermediate object allowing the user to reach some object, of arbitrary nature, its *value*. Here, `*package*` is a symbol, the value of which being the package of name `"COMMON-LISP-USER"`. Technically, a symbol contains a pointer to its possible value, here this package, the package ordinarily used in a simple Lisp session. No means of *directly* citing a package. You can use his *name* and the function `find-package`:

```
> (find-package "COMMON-LISP-USER") ✠
#<The COMMON-LISP-USER package>
```

Conversely, given a package, the function `package-name` returns its name:

```
> (package-name (find-package "COMMON-LISP-USER")) ✠
"COMMON-LISP-USER"
```

Example given to illustrate the functions `find-package` and `package-name` are inverse of each other, defining a 1-1 correspondence between the packages *currently* defined in the environment and their respective names, some character strings.

Because of rules not interesting here, it is common and convenient to use only uppercase character strings to name packages. If a name does not correspond to any current package, the `find-package` function returns the symbol `nil`, usually displayed uppercase:

```
> (find-package "COMMON-LISP-user") ✠
NIL
```

which illustrates that a package-name is case dependent. Most often, only a few packages are used, and because of their importance, descriptive names are used. These descriptive names can be a little long, and it is possible to define and use various *nicknames* for these names. For example, the `"COMMON-LISP-USER"` package usually has the nicknames `"CL-USER"` and `"USER"`. We will always quote the package of name `"COMMON-LISP-USER"` simply as the package `"USER"`.

```
> (find-package "USER") ✠
#<The COMMON-LISP-USER package>
> (package-nicknames (find-package "USER")) ✠
("CL-USER" "USER")
```

Two different packages have disjoint sets of *name + nicknames*, necessary if a package is to be unambiguously identified by its name or one of its nicknames.

4 Tracking a symbol.

The Lisp symbols are very particular with respect to the identifiers of other programming languages. They continue to live during *running time* and are an important component of the power of Lisp. A package is cited via its name, the same for a symbol, but the process is much more complex. The first method, the most common one, to create a symbol, it is better to say "to *allocate* a symbol", consists in writing it using the standard Lisp conventions. In an environment, a set of symbols are defined and can be used. Just to warn the reader about the complexity of the subject, we give this example:

```
> (find-symbol "X" "USER") ✠
NIL
NIL
> x ✠
Error: Attempt to take the value of the unbound variable ‘X’.
[condition type: UNBOUND-VARIABLE]
> (find-symbol "X" "USER") ✠
X
:INTERNAL
```

The function `find-symbol` is analogous to the function `find-package`: given some *name*, a character string, it looks for a symbol having this name. Remember the correspondence between packages and names; the same kind of correspondence between symbols and names, but more sophisticated. In particular, a name is *never sufficient* to determine a symbol, a *package* is also necessary; this is why the `find-symbol` function requires *two* arguments, the name of the symbol and the package with respect to which the search is to be done, this package being determined via its name. At the beginning of our story, no symbol of name `"X"` in the default package `"USER"`, and the answer of `find-symbol` is negative, `nil`. We will explain the second value `nil` later. Then the user gives Lisp the statement in one character '`x`'. Lisp works according to a cycle called *read-eval-print*. First Lisp *reads* the statement, observes the symbol `x` is used, examines the current package `"USER"`, does not *find* such a symbol in this package, so that before anything else, Lisp *allocates* (creates) this symbol, the right terminology being: "Lisp *interns* a symbol `x` of name `"X"` in the package `"USER"`". To reach this symbol, the pair made of *its* name `"X"` and the (not *its*!!) package `"USER"` are necessary, but it would be difficult to write, each time this symbol is to be used, the expression `(find-symbol "X" "USER")`. So that the designers of Lisp have organized the workspace as follows: if a symbol is read, its character string is capitalized, giving its *name*, and the default package is used. So that the text '`x`' is roughly[1] equivalent to the text `(find-symbol "X" "USER")`, convenient! In the second statement of our example, once Lisp has "understood" the user intends to use the symbol `x`, observing such a symbol is not present in the package `"USER"`, it allocates a symbol of name `"X"` and *interns* it in the package `"USER"`; more explicitly a pointer toward the just allocated symbol is added to the list $InS(p)$ of the internal symbols of the package p.

Note also that the standard Lisp habit is to input the symbols in lowercase letters and Lisp on the contrary outputs them in uppercase. Convenient to distinguish input from output in an interactive session, but requiring some lucidity. In particular, no difference between the *symbols* `nil` and `NIL`, whereas the *strings* `"nil"` and `"NIL"` are different.

```
> (find-symbol "nil" "USER") ✠
NIL
NIL
> (find-symbol "NIL" "USER") ✠
NIL
:INHERITED
```

The erroneous search for a *symbol* of name `"nil"`, a *string*, wrong lowercase letters, returns again a double negative answer, but it is the second one which is meaningful! Look at the second statement with the same first answer, but a different second one. The symbol `NIL` plays also in Lisp the role of the boolean *false*, creating an ambiguity: if the answer is the symbol `NIL`, does this mean the answer is negative, or does this mean it is the positive answer made of the symbol `NIL`? The ambiguity is resolved by the *second* answer. In the first case, the second

[1]'x' may allocate a symbol, `find-symbol` never allocates a symbol; see the function `intern` later.

answer `NIL` *confirms* the first answer is to be interpreted as the boolean *false*; in the second statement, the second answer `:INHERITED`, not negative, explains the symbol `NIL` does have been found and that its *accessibility* via the package `"USER"` is *inherited*, a point to be explained later. In other words the function `find-symbol` returns a genuine symbol *and* its accessibility via the package argument, or a pair of `nil`'s if no symbol is found. The logician can deduce an accessibility cannot be the symbol `nil`!

Let us go back to the one character statement '`x`'. The read part of the read-eval-loop cycle reads the symbol `x` and interns it in the package `"USER"`. The statement is *read* and must now be *evaluated.* The evaluation of a symbol consists in looking for a *value* of this symbol; a value pointer in the symbol is examined and in this case some specific pointer indicates that, at this time, no value at all for the symbol `x` just allocated. Therefore the *eval* step of the *read-eval-print* cycle generates an error, terminating the cycle by an informative error message. But anyway a symbol of name `"X"` is now present in the package `"USER"`, as checked by the last statement. The accessibility via the package `"USER"` is *internal*, to be explained later.

5 Examining a symbol.

A symbol is a Lisp object by itself relatively complex, and several functions allow the user to study it. Let us examine in detail the symbol `x` allocated in the previous section.

```
> (symbol-name ’x) ✠
"X"
```

The *symbol-name* function returns the *name* of a symbol, it is analogous to the *package-name* function for a package. The Lisp function `symbol-name` is crucial in our subject and we mathematically denote it as a function $sn : Sm \to St$, a function in general not at all injective: several symbols may have the same name, this is our main subject.

Note the symbol `x` has been quoted `’x` in the statement above to forbid its evaluation. More precisely, `’x` is an abbreviation for the expression `(quote x)` where `quote` is a *special* function which *does not* evaluate its argument before using it, and returns this argument as such. So that the previous statement is equivalent to:

```
> (symbol-name (quote x)) ✠
"X"
```

Here, the non-quoted `x` does not generate an error. If instead you do not quote the symbol `x`, implicitly via `’`, or explicitly via `quote`, it will be evaluated before being used, generating an error:

```
> (symbol-name x) ✠
Error: Attempt to take the value of the unbound variable ‘X’.
[condition type: UNBOUND-VARIABLE]
```

The package which *owns* a symbol is determined by the function `symbol-package`:

```
> (symbol-package 'x) ✠
#<The COMMON-LISP-USER package>
> (find-symbol "X" "USER") ✠
X
:INTERNAL
```

A symbol at least produces a name and a package, and conversely, given a name and a package, `find-symbol` produces the symbol with this name *via* this package. The reader could think here there is a correspondence:

$$Sm \longleftrightarrow St \times P$$

The situation is complicated: for the beginner, it is a convenient point of view, but in fact such a point of view is wrong. If you intend to really understand the complex interconnection between strings, packages and symbols, please forget this tempting correspondence, terribly misleading. The right point of view is the goal of the next sections. We finish this section with the other data possibly stored in a symbol, just to be complete about the nature of a Lisp symbol. A symbol can have a value. You remember in our Lisp session, the symbol `x` does not have a value, which could be safely tested by the predicate `boundp`:

```
> (boundp 'x) ✠
NIL
```

The main method to give a value to a symbol is the `setf` function[2]:

```
> (setf x '(4 5 6)) ✠
(4 5 6)
> (boundp 'x) ✠
T
```

Note the symbol `x` first argument of `setf` is not evaluated, otherwise an error would be generated: at this time the symbol `x` does not yet have a value. Once the `(setf ...)` statement is evaluated, the symbol `x` does have a value, in this case, the list `(4 5 6)`. This value would now be the result of the evaluation of the symbol `x`, which value can also be obtained by the function `symbol-value`[3]. Try to understand why its argument is quoted, the last statement could help.

[2]In fact a macro.

[3]We do not study in these notes the possible difference between on the one hand the simple *evaluation* of the symbol `x` asked by the statement ‘`x`’ and on the other hand by the *evaluation*

```
> x ✠
(4 5 6)
> (symbol-value 'x) ✠
(4 5 6)
> (symbol-value x) ✠
Error: Attempt to take the value of (4 5 6) which is not a symbol.
[condition type: TYPE-ERROR]
```

A symbol can also have a *functional value.* This does not mean the value we just spoke about could be functional, this means that *besides* the previous value, a symbol can also have a *further* value, a function, which can be used independently of the "ordinary" value of this symbol. For example the function $n \mapsto 10-n$ could be recorded as the *functional* value of the same symbol, and this function could then be used as below.

```
> (defun x (n) (- 10 n)) ✠
X
> (x 3) ✠
7
```

Both values coexist in the symbol without any interference:

```
> (symbol-value 'x) ✠
(4 5 6)
> (symbol-function 'x) ✠
#<Interpreted Function X>
```

and can be used independently:

```
> (x (first x)) ✠
6
```

You could already suspect that the position of `x` just after a left parenthesis `(x ...)` makes Lisp extract the *functional* value and not the "ordinary" value. This is correct but is not the subject of these notes. Also a symbol can have a functional value and not an ordinary value; important also but not the subject of this text, an ordinary value can be a functional object, usually then called via `funcall` or `apply`...

A symbol also contains a `plist` with many possible uses, not at all studied here.

of `(symbol-value 'x)`; no difference in our elementary examples; possible differences come from possible different scopes, *lexical* or *dynamic*, a subject not studied here.

6 But what the hell is a package?

We attack now the heart of our subject.

Fact 1: A package $p \in P$ maintains four fundamental lists:

- A list $InS(p)$ of its *internal* symbols.
- A list $ExS(p)$ of its *external* symbols.
- A list $ShgS(p)$ of its *shadowing* symbols.
- A list $U(p)$ of its *used* packages.

First important fact to understand the subject, the first three lists are lists of *symbols*, more precisely, lists of pointers (machine addresses) toward the aimed symbols. Every symbol has a *unique* “existence”, a symbol is a unique machine object in the environment, but *several* packages may *see* the same symbol, that is, include the machine address of this symbol. In particular these lists are not lists of symbol names (strings), assuming another process allows Lisp to reach a symbol when its name is known. Every symbol is directly accessible via a package appropriate for some reason, and several packages can in general play this role at a moment of a Lisp session for the same symbol. In particular, for example, all the standard Lisp symbols locating the main functions of the Lisp architecture are in general accessible via any package.

No duplicate symbol, more precisely, no duplicate pointer in each of these lists. Also the relations $Ins(p) \cap ExS(p) = \emptyset$ and $ShgS(p) \subset InS(p) \cup ExS(p)$ must be satisfied, for reasons studied later.

Fact 2: Given the current set of packages, assumed *coherent,* given the current state of all the fundamental lists of these packages, the function `find-symbol` can unambiguously determine *whether* a symbol of name some string is *accessible* via some package; if so, this symbol is unique.

The importance of the package argument of `find-symbol` could make the reader believe the function `symbol-package` will be useful for various tests. Not at all: this function returns only the “birth place” of the symbol, technically called the *home package* of this symbol, but which says almost nothing about the current status of this symbol with respect to any package; says only it is at least *interned* in this package. In particular a symbol can be accessible via several packages, possibly different from its home package. The simplest illustration:

```
> (symbol-package 'nil) ✠
#<The COMMON-LISP package>
> (find-symbol "NIL" "COMMON-LISP") ✠
NIL
:EXTERNAL
> (find-symbol "NIL" "USER") ✠
NIL
:INHERITED
> (eq (find-symbol "NIL" "COMMON-LISP")
      (find-symbol "NIL" "USER")) ✠
T
```

The *home package* of `nil` is `"COMMON-LISP"` and this symbol of name `"NIL"` is accessible via `"COMMON-LISP"`, but also via `"USER"`. Not with the same status, *external* in the first case, *inherited* in the second case. Points to be examined later. The last statement *proves* both symbols determined by `find-symbol` via different packages in fact are the same. We can make a little more precise the Fact 2. At some time of a Lisp session, a set P of packages are defined and also a set Sm of symbols. The various lists of symbols $InS(p)$, $ExS(p)$ and $ShgS(p)$, and the list of packages $U(p)$ of a package $p \in P$ define without any ambiguity which symbols are accessible via the package p. In other words, a well defined function is defined:

$$\rho : P \to [St \to Sm \cup \{\texttt{nil}\}]$$

We choose the letter ρ for reader: this function is mainly used by the reader. Our "mathematical" function ρ is nothing but an avatar of the Lisp function `find-symbol`. Let $p \in P$ be a package and $st \in St$ be a string, then:

$$\rho(p)(st) = (\texttt{find-symbol}\ st\ p)$$

The lefthand member uses the standard mathematical functional notation, and the righthand one the Lisp notation. If no symbol is found for a pair (st, p), the symbol (!) `nil` is returned and the second value of `find-symbol`, the symbol `nil` again, allows the user to understand the symbol search failed. A package p is nothing but a process defining the function $\rho(p) : St \to Sm \cup \{\texttt{nil}\}$. In other words, the four fundamental lists of a package determine what character strings are the names of the symbols accessible via this package. According to a method to be detailed now.

7 Internal symbols.

The first fundamental list of a package is the list of its *internal symbols*. It is a pity no standard Lisp function can produce this list. A solution is this one:

```
> (defun package-internal-symbols (package)
    (let ((rslt nil))
      (do-symbols (s package)
        (when (eq (second
                    (multiple-value-list
                      (find-symbol (symbol-name s) package)))
                  :internal)
          (push s rslt)))
      rslt)) ✠
PACKAGE-INTERNAL-SYMBOLS
```

We can list now all the internal symbols of a package:

```
> (package-internal-symbols "USER") ✠
(ICONNECTIONPOINTCONTAINER ICONNECTIONPOINT RFE6406 IDISPATCH
 PACKAGE-INTERNAL-SYMBOLS S SET-APPLICATION-ICON BUNDLE
 SET-DEFAULT-COMMAND-LINE-ARGUMENTS RSLT ...)
```

In fact only the first ten internal symbols are displayed. We see in particular the symbols `package-internal-symbols`, `s` and `rslt` used in the definition of our function. Some are rather esoteric, strictly speaking illegal: at the beginning of a Lisp session, the `"USER"` package should in principle be void of symbols. It is better to allocate and use toy packages to make more understandable our experiments. The `defpackage` function, in fact a macro, allocates a package of a given name.

```
> (defpackage "P1") ✠
#<The P1 package>
```

A package `"P1"` is now present in our environment, without any symbols.

```
> (package-internal-symbols "P1") ✠
NIL
```

The symbol `nil` is also the void list. We intend to allocate some symbols and make them internal in the package `"P1"`. The current default package is `"USER"` and we may switch to `"P1"`:

```
> (in-package "P1") ✠
#<The P1 package>
> *package* ✠
#<The P1 package>
```

The default package is from now on `"P1"`. If we write now a symbol, except if explicitly expressed differently, it will be understood as accessed via the package `"P1"`; let us allocate a list of three symbols assigned to the symbol `list`, and then examine the internal symbols of the package `"P1"`:

```
> (setf list '(symb1 symb2 symb3)) ✠
(SYMB1 SYMB2 SYMB3)
> (package-internal-symbols "P1") ✠
Error: attempt to call ‘PACKAGE-INTERNAL-SYMBOLS’ which is an undefined function.
[condition type: UNDEFINED-FUNCTION]
```

Oops! What happens? We successfully used in the previous page our function `package-internal-symbols` and suddenly this function seems absent of the environment. The point is that it was *defined when* `"USER"` was the default package. Let us examine the situation.

```
> (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "USER") ✠
COMMON-LISP-USER::PACKAGE-INTERNAL-SYMBOLS
:INTERNAL
> (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "P1") ✠
PACKAGE-INTERNAL-SYMBOLS
:INTERNAL
```

A symbol with the given name is found in both packages; before meeting the error of functional value not defined, Lisp, more precisely the Lisp reader had interned a *new* symbol also of name `"PACKAGE-INTERNAL-SYMBOLS"` in the current default package `"P1"`, for the only reason it was present in the source text. Let us check both symbols are different:

```
> (eq (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "USER")
      (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "P1")) ✠
NIL
```

These symbols are not the same, compare with the status of `nil`, where the *same* symbol `nil` was accessible via `"LISP"` and via `"USER"` as well. In our new situation, both symbols are *different*, and also their respective properties. For example the first one does have a functional value, the second one not, which can be tested by the function `fboundp`:

```
> (fboundp (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "USER")) ✠
T
> (fboundp (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "P1")) ✠
NIL
```

For the first time, we have seen the double-colon '`::`' notation. When a symbol was asked via `"USER"`, the answer was positive but the returned symbol was `COMMON-LISP-USER::PACKAGE-INTERNAL-SYMBOLS`, a little lengthy. The current default package is `"P1"`, via which *this* symbol is not accessible, but we may however access it using the double-colon notation:

```
package-name::symbol-name
```

This is one of the methods allowing the user, or Lisp, to cite a symbol not accessible via the default package; the reader is told: "We mean the symbol `symbol-name` via the package `package-name`". On the contrary, for the symbol with the same name, but interned in the different package `"P1"`, because `"P1"` is the current default package, the double colon notation is not necessary. This allows us to use the first symbol interned in the `"USER"` package and its functional value, so accessible:

```
> (user::package-internal-symbols "P1") ✠
(PACKAGE-INTERNAL-SYMBOLS SYMB1 SYMB3 SYMB2)
```

The `user::` prefix tells Lisp to look for the symbol via the `"USER"` package, you remember `"USER"` is a nickname for `"COMMON-LISP-USER"`, and because of its position just after a left parenthesis, to use its functional value. Currently four internal symbols in the package `"P1"`, delivered in an arbitrary order. By the way, what about the symbol `list`, to which our list (!) of symbols is assigned?

```
> (find-symbol "LIST" "P1") ✠
LIST
:INHERITED
```

The symbol is accessible via the package `"P1"` but with a different status, the status *inherited*. We will examine this other situation later. In fact, the symbol `list`, one of the most important in Lisp, is already present in the package `"COMMON-LISP"` at the beginning of a session, and by default accessible from any package.

We must mention also that our function ρ, or if you prefer the function `find-symbol`, is (almost) injective *with respect* to *st*. Let p_1 and p_2 be two packages, st_1 and st_2 two strings. Then if $\rho(p_1, st_1) = \rho(p_2, st_2) \neq \texttt{nil}$, then $st_1 = st_2$, pay attention to the fact the equality is an equality between *symbols*. This is a consequence of `symbol-name`$(\rho(p, st)) = st$; in other words, no possible nickname for a symbol name. In particular, a 1-1 correspondence is defined between the internal symbols of a package and their respective names.

8 Importing a symbol.

Instead of using the heavy notation `user::xxx` to make accessible from `"P1"` a symbol internal in `"USER"`, if we judge our function `package-internal-symbols` deserves an importation, we may decide to *import* this symbol in the package `"P1"`. But we meet another problem due to a name conflict, a point examined in detail later:

```
> (import 'user::package-internal-symbols "P1") ✠
Error: Importing these symbols into the P1 package causes a name conflict:
(COMMON-LISP-USER::PACKAGE-INTERNAL-SYMBOLS)
[condition type: PACKAGE-ERROR]
```

The problem is a "collision" between two *different* symbols: both cannot coexist as internal symbols in the *same* package `"P1"` with the *same* name. Importing the symbol internal in `"USER"` into the package `"P1"` is not compatible with the current presence of *another* symbol in `"P1"` with the same symbol-name. We must first `unintern` the annoying symbol pre-existing in `"P1"` to let a "free space" to the symbol of `"USER"`.

```
P1(15): (unintern 'package-internal-symbols "P1") ✠
T
P1(16): (import 'user::package-internal-symbols "P1") ✠
T
> (package-internal-symbols "P1") ✠
(PACKAGE-INTERNAL-SYMBOLS SYMB1 SYMB3 SYMB2)
```

The Lisp terminology is rather misleading: in fact it is not at all the symbol which is imported, it is the connection *string* $\mapsto$ *symbol*. More precisely, let p_1 and p_2 be two packages, *st* a string and *sm* a symbol; if $\rho(p_1)(st) = sm$ before the importation, then after the importation, both relations $\rho(p_1)(st) = sm$ *and* $\rho(p_2)(st) = sm$ are satisfied; in particular the first relation remains valid, it is not really an importation, it is a *copy* in the package p_2 of the connection $st \mapsto sm$, available in the package p_1.

In particular nothing is modified for the symbol itself, except *possibly*, rarely, for its home package, as examined later. It is tempting for a beginner to think the home-package of the symbol is just modified, not at all! Another erroneous interpretation would be a *copy* of the symbol to be imported “inside” the target package, with all the corresponding ingredients of the symbol, value, functional value, and so on, not at all either! It is really a copy which is done, but only of a *pointer* toward our symbol, found in $InS(p_1)$, copied in $InS(p_2)$.

The `import` function has two arguments, the *symbol* to be imported and the *target package*. The symbol must be well defined, that is, taking account of the current state of the environment. Here, before importing, the symbol is accessible only via `"USER"` and it is mandatory to use the notation `user::...`; also the symbol must be quoted to preclude its evaluation. You see the symbol can then be functionnally used from the package `"P1"` without using the prefix notation. Also the imported symbol is in the list of the internal symbols of the package `"P1"`, but it is not the same as before. The next mandatory exercise consists in comparing the original symbol and the imported one.

```
> (eq 'user::package-internal-symbols ✠
      'package-internal-symbols)
T
```

The function `eq` proves both notations denote the same symbol. Other proof:

```
> (eq (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "USER") ✠
      (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "P1"))
T
```

If ever you suddenly remember the existence of the `symbol-package` function, the next test is tempting:

```
> (symbol-package (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "USER"))
#<The COMMON-LISP-USER package>
> (symbol-package (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "P1"))
#<The COMMON-LISP-USER package>
```

The *unique* symbol accessed by two different paths has of course a unique home package, namely its *birth place*, the `"USER"` package. Please, do not pay any attention to this symbol package; it is just a matter of history and does not have any real interest. You must just focus on the chain *package* $\rightarrow$ [*string* $\rightarrow$ *symbol*]. No reason to be interested by the map `symbol-package` : *symbol* $\rightarrow$ *package*.

9 Name conflicts.

The possible occurence of *name conflicts* is essential when organizing packaging. The lovely organization of the Lisp packaging system requires to understand an essential point, without which you can be far from the right judgment.

Warning. A name conflict can only be generated by two *different* symbols $sm, sm' \in Sm$ having the *same* name: $sn(sm) = sn(sm')$; a name conflict happens if these symbols sm and sm' become accessible via the same package p, a forbidden situation, making ambiguous the `find-symbol` function. On the contrary, the fact some symbol (singular) is accessible via *different* packages never generates a name conflict by itself.

Lisp constantly watches possible name conflicts and stops on error if some statement causes a name conflict. Some errors can be *continuable*: this means Lisp is then able to propose solutions, via `unintern` or `shadowing-import`, to resolve the conflict. We explicitly used `unintern` for this reason in the previous section.

What happens if an imported symbol has the same *name* as a symbol already accessible in the target package. In mathematical language, let p_1 and p_2 be two different packages, st some string, sm_1 and sm_2 two *different* symbols respectively internal in the packages p_1 and p_2 with the same name st. In other words, $\rho(p_1)(st) = sm_1$ and $\rho(p_2)(st) = sm_2$; also $sm_1 \in InS(p_1)$ and $sm_2 \in InS(p_2)$. We assume $sm_1 \neq sm_2$. An initial situation of this sort could be the following, remember the current default package is `"P1"`:

```
> (setf user::symb4 111) ✠
111
> (setf symb4 111) ✠
111
> (eq 'user::symb4 'symb4) ✠
NIL
```

The first statement interns a symbol of name `"SYMB4"` in the package `"USER"` and gives it the value 111, the second statement interns a *different* symbol again of name `"SYMB4"` in the package `"P1"`, the current default package, and also gives it the value 111. Two symbols have been allocated, installed somewhere in the environment, with their respective values 111 which happen to be the same; also the connections $\rho($`"USER"`$)$: `"SYMB4"` $\mapsto$ `user::symb4` and $\rho($`"P1"`$)$: `"SYMB4"` $\mapsto$ `p1::symb4`, have been installed, that is, the addresses of both symbols have been pushed in the *respective* lists of *internal* symbols of the packages `"USER"` and `"P1"`. The `eq` comparaison proves the just allocated symbols are different. Two symbols with the same name `"SYMB4"` are now present in our environment; this is possible because these symbols are interned in different packages. Each one is accessible via the appropriate package. Both symbols are alive and can be freely used:

```
> (setf user::symb4 (+ symb4 user::symb4)) ✠
222
```

What happens if we import `user::symb4` into the package `"P1"`? Test:

```
> (import 'user::symb4 "P1") ✠
Error: Importing these symbols into the P1 package causes a name conflict:
(COMMON-LISP-USER::SYMB4)
[condition type: PACKAGE-ERROR]
```

The importation is rejected, for it would not be anymore possible to coherently define ρ(`"P1"`)(`"SYMB4"`). We will see later that another process named *shadowing* allows the user to put away the symbol previously present in `"P1"` to authorize the importation of the symbol present in `"USER"`.

10 The function `intern`.

Fact 3: Let p be a package and $(sm_1, \ldots, sm_n)$ the list of its internal symbols. If p is the default package, any of these symbols can be directly used without the double colon notation '`::`'. If p is not the default package, these symbols can be used thanks to the notation p`::`sm_i. All these internal symbols have different names. Nothing prevents several packages to have the same symbol in their list of internal symbols. This is possible thanks to the *import* function which can work only if no name conflict is generated.

To be complete on this matter, we are ready now to consider the Lisp function `intern`. It is used almost exactly as the function `find-symbol` with just a difference: if no symbol is found, then a symbol is allocated and interned in the package argument. Compare:

```
> (find-symbol "SYMB5" "P1") ✠
NIL
NIL
> (intern "SYMB5" "P1") ✠
SYMB5
NIL
> (intern "SYMB5" "P1") ✠
SYMB5
:INTERNAL
```

We start without any `symb5` symbol in `"P1"`, checked by the double negative answer of `find-symbol`. If instead we use the function `intern`, Lisp observes such a symbol is absent in `"P1"` and therefore allocates a symbol of name `"SYMB5"` and *interns* it in the package `"P1"` as an internal symbol; the second value `nil` is a witness of the just made allocation. If we repeat the same statement, it is not an error, but Lisp signals such a symbol is *already* present and informs of its status, which also informs of its previous presence. A little weird! This definition of the function `intern` guarantees the coherence of the set of internal symbols of packages. A symbol can be allocated only through this function `intern`. Two arguments, a string *st* and a package *p*. In a sense the function `intern` first calls `(find-symbol` *st p*`)` with the same arguments; if a symbol is found it is returned without modification, no allocation of a new symbol, which would certainly induce a name conflict. On the contrary, if a symbol is not found corresponding to this definition, it is allocated and interned in the package explicitly or implicitly quoted, and this new symbol certainly does not generate a name conflict. In particular, in this way, the internal symbols of a package have certainly respective names different from each other. On the contrary, two different symbols of the environment can have the same names, but the packaging system is organized such that they cannot be accessed via the same package.

```
> (package-internal-symbols "P1") ✠
(SYMB5 PACKAGE-INTERNAL-SYMBOLS SYMB4 SYMB1 SYMB3 SYMB2)
```

In fact the Lisp reader constantly uses the `intern` function. Reading a statement, if it observes the presence of a symbol, the colon notation allows it to decide if a package is explicitly given; if not, it must use the default package. The name of the symbol, alone or following the colon notation, then determines the symbol; if present it is used, else it is allocated and then used. It is exactly the role of the `intern` function.

11 Moving a symbol between two packages.

Up to now, no means of modifying the home package of a symbol. In fact, it is possible but a little contorted. Almost without any interest except the opportunity to use this theme to learn other facts. First, it is possible to *unintern* a symbol. For example we can unintern the symbol `"SYMB1"` from the package `"P1"`. Precisely, this symbol is removed from the list of its internal symbols.

```
> (unintern 'symb1 "P1") ✠
T
> (package-internal-symbols "P1") ✠
(SYMB5 PACKAGE-INTERNAL-SYMBOLS SYMB4 SYMB3 SYMB2)
```

But which is modified is in fact the *list* of internal symbols in the package `"P1"`. The symbol `symb1` itself has not been modified, except possibly its home package; in particular the symbol is not destroyed, therefore always present somewhere in the environment. But inaccessible as such, for a symbol most often is accessible only via a package, the default one or some other explicitly given. Our symbol `symb1` was interned only in the package `"P1"`, so that it seems now unreachable! If a symbol is really unreachable, the Lisp garbage collector, always silently working below your session, is able to prove it; then it frees the corresponding part of memory for a later possible use by something else. But here, our symbol remains reachable! for it was the first element of the list assigned to the symbol `list`, so that it is interesting to look at its value:

```
> list ✠
(#:SYMB1 SYMB2 SYMB3)
```

Our symbol is still present in the list but under a form a little different. It is still a symbol:

```
> (type-of (first list)) ✠
SYMBOL
```

But what about its home package?

```
> (symbol-package (first list))
NIL
```

No home package, our symbol is homeless! You understand the prefix '`#:`' means this symbol is homeless. This has been done by `unintern` because the package argument of `unintern` was the package which *owned* the symbol; otherwise the home package is not modified. Our symbol is become homeless but can continue to live as an ordinary symbol, for example we can assign it a value:

```
> (set (first list) 444) ✠
444
> (symbol-value (first list)) ✠
444
> (eval (first list)) ✠
444
```

Unlike `setf`, the `set` function evaluates its arguments, so that before assigning, the expression `(first list)` is evaluated, returning our mysterious symbol `#:symb1`; this done, the `set` function assigns (the value of) the second argument to it. Checked by the other statements. It is even possible to *import* a homeless symbol in a package, but then the target package becomes the home package of the symbol; which package can be different from the original one: our symbol is "moved" into another package. Which explains the new form of the value of `list`.

```
> (import (first list) "USER") ✠
T
> (symbol-package 'user::symb1) ✠
#<The COMMON-LISP-USER package>
> list
(COMMON-LISP-USER::SYMB1 SYMB2 SYMB3)
```

Our symbol could also have been imported in other packages before being uninterned from his home package. The status for our symbol with respect to other packages would not have been modified.

```
> (import 'symb3 "USER") ✠
T
> (eq 'user::symb3 'symb3) ✠
T
> (unintern 'symb3 "P1") ✠
T
> (eq 'user::symb3 (third list)) ✠
T
> (symbol-package 'user::symb3) ✠
NIL
> (import 'user::symb3 "USER") ✠
T
> (symbol-package 'user::symb3) ✠
NIL
```

Here we import `symb3` in the package `"USER"` *before* uninterning it from the package `"P1"`. The symbol is then accessible via the package `"USER"` or via the list `list`; The `(eq ...)` statement proves both paths go to the same address. When `import` works, the symbol does have a home package, so that it is not changed. When it is uninterned from this package, it becomes homeless; Lisp does not take account of the fact it is internal in other packages, which could allow it to choose one of these packages as an "emergency" package, but which one? Furthermore this would force Lisp to examine the list of all the other packages and all their accessible symbols: too expensive! Here, even if the symbol is internal and therefore interned in the package `"USER"`, it becomes homeless after the `unintern`. Finally we import it in the package... `"USER"`, where in fact *it is already present*: in such a situation, its status is unchanged, it is the rule, in particular its home package remains `nil`. A little esoteric? Don't be afraid: the home package does not have any real role for the Lisp user.

We will *prove* later that if a symbol *sm* has a home package p, then *sm* is certainly *present* in p, that is, internal or external in this package. You see the "converse" is false: a symbol may be present in several packages $p_1, \ldots, p_k$ and yet not having a home package. You could feel a little weird about such a fact, but the best is not to take account of it: except fot the internal efficiency of the Lisp interpeter and compiler, the notion of home-package is almost useless for the programmer.

12 External symbols and used packages.

Analyzing this notion of internal symbol was a little long, but in this way, the general structure of the packaging system is now clear; it will so be easier to understand the other notions. Two fundamental lists in the packages play a "dual" role in the packaging system, the list of *external* symbols and the list of *used* packages. The following function gives the list of the external symbols of a package.

```
> *package* ✠
#<The P1 package>
> (defun package-external-symbols (package) ✠
    (let ((rslt nil))
      (do-external-symbols (s package)
        (push s rslt))
      rslt))
PACKAGE-EXTERNAL-SYMBOLS
```

The first statement just to inform the default package remains `"P1"`. Let us see what the lists of external symbols of the packages `"P1"`, `"USER"` and `"LISP"` are:

```
> (package-external-symbols "P1") ✠
NIL
> (package-external-symbols "USER") ✠
NIL
> (package-external-symbols "LISP") ✠
(LOGTEST CONSP BREAK CDDR DEFGENERIC +++ ARRAY-DIMENSION READTABLEP
 SETQ WRITE-TO-STRING ...)
> (length *) ✠
978
```

No external symbol in `"P1"` and `"USER"`, but 978 such symbols in the package `"LISP"`, only the first ten ones being displayed. A point could generate some trouble: these symbols are accessible via the package `"LISP"` and yet the notation `lisp::logtest` for example has not been used. The explanation has two *dual* components:

1. The symbol `logtest` is *external* in the package `"LISP"`.
2. The package `"LISP"` is in the `package-use-list` of the package `"P1"`:

```
> (find-symbol "LOGTEST" "LISP") ✠
LOGTEST
:EXTERNAL
>(package-use-list "P1") ✠
(#<The COMMON-LISP package>)
```

`"LISP"` is the common nickname of the package whose official name is `"COMMON-LISP"`, not to be confused with `"COMMON-LISP-USER"`.

```
> (eq (find-package "LISP") (find-package "COMMON-LISP")) ✠
T
> (eq (print (find-package "LISP"))
      (print (find-package "USER"))) ✠
#<The COMMON-LISP package>
#<The COMMON-LISP-USER package>
NIL
```

The Lisp function `package-use-list` applied to a package p returns the list of all the packages used by p. If p_1 uses p_2, then it is said p_2 is used by p_1. This relation is neither reflexive, most often p does not use p, nor symmetric: "p_1 uses p_2" does not imply "p_2 uses p_1", nor transitive: "p_1 uses p_2" and "p_2 uses p_3" does not imply "p_1 uses p_3". The oriented graph of the *use* relation is totally arbitrary.

13 First toy examples.

It is simpler to restart a new Lisp session from scratch where we allocate two packages `"P1"` and `"P2"`.

```
> (defpackage "P1") ✠
#<The P1 package>
> (defpackage "P2") ✠
#<The P2 package>
```

We switch the default package to `"P1"`:

```
> (in-package "P1") ✠
#<The P1 package>
```

and we assume that, *in this environment*, we define the functions `package-internal-symbols` and `package-external-symbols` as explained before. What about the internal symbols of `"P1"`:

```
> (package-internal-symbols "P1") ✠
(RSLT PACKAGE-INTERNAL-SYMBOLS PACKAGE-EXTERNAL-SYMBOLS S)
```

We see the symbols locating our functions and also the symbols `rslt` and `s` used to define them. All the other symbols used in their definitions are in fact external in `"LISP"` and absent here. We move now to the package `"P2"` where we cannot access directly from this package to the symbol `package-internal-symbols`:

```
> (in-package "P2") ✠
#<The P2 package>
> (find-symbol "PACKAGE-INTERNAL-SYMBOLS" "P2") ✠
NIL
NIL
```

We have already seen the solution consisting in using the '`::`' notation allowing the user to reach *one* time the symbol. We can also *import* it, in which case it becomes definitely directly accessible:

```
> (import 'p1::package-internal-symbols "P2") ✠
T
> (package-internal-symbols "P2") ✠
(PACKAGE-INTERNAL-SYMBOLS)
```

Another solution needs *two* actions:

1. Make the symbol *external* in some package, this is the role of the Lisp function `export`.
2. Declare that the current package *uses* the package where the symbol is external.

Let us use this method to make directly visible from `"P2"` the symbol `package-external-symbols`.

```
> (export 'p1::package-external-symbols "P1") ✠
T
> (use-package "P1" "P2") ✠
T
> (package-external-symbols "P2") ✠
NIL
> (package-external-symbols "P1") ✠
(PACKAGE-EXTERNAL-SYMBOLS)
```

An `export` statement has the same form as an `import` one: `(export` *sm st*`)` with *sm* a symbol and *st* a string naming a package. The symbol *sm* must be readable from the current default package, for us `"P2"`, explaining the required prefix '`p1::`'; the argument *st* is a string naming the package where the symbol *sm* is ordered to become external. Again, the terminology is somewhat misleading: the function `export` does not export anything, it makes only some connection $st \to sm$ "exportable" to other packages via `use-package` statements. The 978 external symbols of `"LISP"` contain all the standard Lisp constants, variables and functions. Because an allocated package is always initialized with the package `"LISP"` in its `package-use-list`, these basic Lisp objects will be accessible from any package. If we remove `"LISP"` from the `use-package-list` of the current default package, possible with `unuse-package`, the situation becomes difficult.

```
> (unuse-package "LISP" "P2") ✠
COMMON-LISP:T
> (+ 2 2) ✠
Error: attempt to call '+' which is an undefined function.
[condition type: UNDEFINED-FUNCTION]
```

The notation for the external symbol `t` in the package `"LISP"` is become `common-lisp:t`, for `"LISP"` is no more used by `"P2"`; the symbol `t` being external in `"LISP"`, the simple colon notation is sufficient. The most elementary operations are not anymore ordinarily reachable, for example the addition function '`+`' now is not directly accessible. Not sensible, let us go back to the previous environment.

```
P2(20): (use-package "LISP" "P2")
Error: attempt to call 'USE-PACKAGE' which is an undefined function.
[condition type: UNDEFINED-FUNCTION]
```

Does not work: the symbol `use-package` is no more accessible either, we *must* use the prefix notation.

```
P2(21): (lisp:use-package "LISP" "P2")
Error: Using package ‘COMMON-LISP’ results in name conflicts for these symbols:
       USE-PACKAGE +
[condition type: PACKAGE-ERROR]
```

But a new problem: previously, when trying to directly use the symbol ‘`+`’ from `"P2"`, Lisp observed such a symbol is not accessible from `"P2"` and therefore interned a fresh symbol of name `"+"` in `"P2"`. This *was* legal but it is not *now* compatible with `"LISP"` having a *different* symbol with the same name `"+"` if `"P2"` uses `"LISP"`. The same for... `use-package`! We must unintern these symbols from `"P2"` to resolve this conflict, without forgetting to use the colon notation for `unintern`, otherwise unreachable.

```
P2(30): (lisp:unintern ’+ "P2") ✠
COMMON-LISP:T
P2(30): (lisp:unintern ’use-package "P2") ✠
COMMON-LISP:T
```

Allowing another try:

```
> (lisp:use-package "LISP" "P2") ✠
T
> (+ 3 4) ✠
7
```

We are back in the standard situation where Lisp is used.

14 Importing an external symbol.

Remember a symbol *sm* is *accessible* from a package p if $\rho(p)(st) = sm$ for $st =$ `symbol-name`(sm). This is possible if *sm* is one of the internal symbols of p, in which case `find-symbol` will give `:internal` as a second answer, or if *sm* is one of the external symbols of p, in which case `find-symbol` will give `:external` as a second answer, and otherwise if *sm* is an external symbol of a package p' used by the package p, in which case `find-symbol` will give `:inherited` as a second answer. We have seen how to import a symbol from a package p' into a package p. The given example used a symbol *sm* internal in the package p', which became also an internal symbol of the package p after the importation. This can be done as well in the same way if *sm* is external in the package p', if p uses p' or not. Of course unless a name conflict is generated. We restart a Lisp session, we allocate the packages `"P1"` and `"P2"`, define the package `"P1"` as the default package and play as follows.

```
> (export (intern "SYMB1" "P2") "P2") ✠
T
> (find-symbol "SYMB1" "P2") ✠
P2:SYMB1
:EXTERNAL
> (find-symbol "SYMB1" "P1") ✠
NIL
NIL
```

We intern a symbol `p2::symb1` in `"P2"` and immediately make it external. This symbol is not accessible via `"P1"`. It is external in `"P2"` but this does not prevent us from importing it in `"P1"`.

```
> (import 'p2:symb1 "P1") ✠
T
> (find-symbol "SYMB1" "P1") ✠
SYMB1
:INTERNAL
> (find-symbol "SYMB1" "P2") ✠
SYMB1
:EXTERNAL
```

Once the importation is done, the symbol is accessible via `"P1"` with the status *internal* and keeps the previous status *external* with respect to `"P2"`. This illustrates in particular the fact that this matter of status of a symbol does not depend on anything in the symbol itself; it is the *package* which must be examined to determine the status of this symbol *with respect to this package*.

Curiously, the last `find-symbol` answers the symbol `symb1`, without the prefix '`p2:`', because Lisp outputs a symbol according to the following rule:

> Lisp first determines whether this symbol is accessible via the current default package, in which case it is output without using a colon notation; otherwise Lisp examines the home package where the symbol is certainly internal or external, in which case it is output with the double or simple colon notation; exceptionally, a *keyword*[4], that is, a symbol of the package `"KEYWORD"`, is output simply '`:xxx`', the prefix `keyword` being in this unique case omitted; finally if the symbol is homeless, it is output `#:xxx`.

The point not to be forgotten is that an importation does not change anything for the relevant symbol, it is just a matter of updating a list of internal symbols in some package, that is, a list of respective pointers toward these symbols. Installing somewhere at San Francisco a new signpost toward New-York does not change anything at New-York. To be sure:

[4]Though the Ansi definition is somewhat uncomplete for the `"KEYWORD"` package, cautiously consider it is forbidden to change anything in the scope of the keywords.

```
> (eq (find-symbol "SYMB1" "P1") (find-symbol "SYMB1" "P2")) ✠
T
```

A novice could believe that adding `"P2"` to the `package-use-list` of `"P1"` could generate a name conflict, because the symbols `p1::symb1` and `p2::symb1` would become both accessible via `"P1"`. Not at all: both symbols are in fact the same, and this situation does not generate any ambiguity.

```
> (use-package "P2" "P1") ✠
T
> (find-symbol "SYMB1" "P1") ✠
SYMB1
:INTERNAL
> (find-symbol "SYMB1" "P2") ✠
SYMB1
:EXTERNAL
```

Note in particular that because our symbol is internal in `"P1"`, the function `find-symbol` gives it the status `:internal`, in a sense preferred to the status `:inherited` which after all would be possible. A name conflict happens only if two *different* symbols with the same name could be accessible via the same package, possibly via another one used by the first one. Two symmetrical illustrations:

```
> (export (intern "SYMB2" "P2") "P2") ✠
T
> (intern "SYMB2" "P1") ✠
SYMB2
:INHERITED
```

No error: in this case, the second `intern` *does not* allocate a new symbol, a symbol of name `"SYMB2"` being already accessible via `"P1"`, which uses `"P2"`; in particular the second `intern` does not import the pre-existing symbol. We must imagine something else to really allocate a new symbol.

```
> (make-symbol "SYMB2") ✠
#:SYMB2
> (import * "P1") ✠
Error: Importing these symbols into the P1 package causes a name conflict:
(#:SYMB2)
[condition type: PACKAGE-ERROR]
```

The function `make-symbol` allocates a *homeless* symbol of given name, it is its definition. Such a homeless symbol cannot generate a name conflict, because it is *not* accessible via any package. Nothing prevent to try to `import` this symbol[5] in `"P1"`; but this generates a name conflict between two different symbols: the fresh

[5]The '`*`' denotes a symbol which always points to the last result returned, here our homeless symbol.

allocated `#:simb2` and the symbol external in `"P2"` accessible also via `"P1"` which uses `"P2"`. Another way to generate a name conflict:

```
> (intern "SYMB3" "P1") ✠
SYMB3
NIL
> (intern "SYMB3" "P2") ✠
P2::SYMB3
NIL
> (export * "P2") ✠
Error: Exporting the symbol P2::SYMB3 from the ‘P2’ package would cause
          some packages that use the ‘P2’ package
          to have two accessible but distinct symbols with the same
          print name.  The packages and conflicting symbols are:
   package ‘P1’:  SYMB3
[condition type: PACKAGE-ERROR]
```

The package `"P1"` uses the package `"P2"`, but this does not prevent from having two different internal symbols with the same name `"SYMB3"` in these packages, for `"P1"` does not see the symbol internal in `"P2"`. If we try to make external this symbol in `"P2"`, a name conflict occurs. Another sort of name conflict can occurs between two used packages:

```
> (export (intern "SYMB4" "P2") "P2") ✠
T
> (export (intern "SYMB4" (defpackage :p3)) "P3") ✠
T
> (use-package "P3" "P1") ✠
Error: Using package ‘P3’ results in name conflicts for these symbols: SYMB4
[condition type: PACKAGE-ERROR]
```

To be compare with:

```
> (export (intern "SYMB5" "P2") "P2") ✠
T
> (export (import ’p2:symb5 "P3")) ✠
T
> (unexport ’p3:symb4 "P3") ✠
T
> (use-package "P3" "P1") ✠
T
```

Two symbols of name `"SYMB5"` are external in `"P2"` and `"P3"` but the fake second one is in fact the same as the first one: it has been imported and immediately made external in `"P3"`. This time the `use-package` making the external symbols of `"P2"` *and* `"P3"` visible from `"P1"` does not generate a name conflict: no name conflict if two different paths lead to the same *symbol*. At least if we do not forget *before* to `unexport` the symbol `"SYMB4"` of `"P3"`, the symbol causing the previous name conflict, that is to downgrade its status from *external* to *internal*.

15 A symbol can shadow another symbol.

In this matter of name conflicts, the Lisp conceptors could have adopted another strategy consisting in deciding that the symbols *present* in a package, that is, *internal* or *external* in this package, have precedence over *inherited* possibly different symbols. No, it was decided this could easily generate true programming errors because of possible perverse collision of identifiers, one of the most terrible sources of bugs. However, according to the general spirit of Lisp, if a programmer finally wants to use such a strategy, he can *exceptionally* do it, explicitly describing what the wished exemptions are. This is the role of the `package-shadowing-symbols` list maintained in every package, denoted by $ShgS(p)$ if the relevant package is p, which lists are initially empty. Such a list can contain symbols present in the concerned package, internal or external; these symbols will automatically *shadow* any other symbol in competition, because exported in other used packages and with the same *symbol-name*.

The function `shadow` quotes a string understood as a symbol-name, allocates a symbol with this name if it is not *present* in this package, and records it in the shadowing list. We continue the same Lisp session.

```
> (shadow "SYMB6" "P1") ✠
T
> (package-shadowing-symbols "P1") ✠
(SYMB6)
> (export (intern "SYMB6" "P2") "P2") ✠
T
> (eq 'symb6 'p2:symb6) ✠
NIL
```

Once the symbol `p1::symb6` is on the shadowing-list of `"P1"` the existence of a *different* external symbol with the same name in the used package `"P2"` does not generate a name conflict. The last `eq`-test proves both symbols are really different.

Along the same line, the function `shadowing-import`, as clearly indicated in its name, imports a symbol and immediately adds it to the shadowing list, which could resolve some possible name conflict. For example the *different* symbols `p2:symb4` (external in `"P2"`) and `p3::symb4` (internal in `"P3"`) are in our environment; the package `"P1"` uses both packages `"P2"` and `"P3"`, but `p3::symb4` is internal in `"P3"` and therefore not in conflict with `p2::symb4`. Remember also the current default package is `"P1"`. A simple importation of `p3::symb4` in `"P1"` generates a conflict, while a `shadowing-import` adds the imported symbol in the shadowing list, shadowing the symbol `p2:symb4` and resolving the name conflict.

```
> (find-symbol "SYMB4" "P2") ✠
SYMB4
:EXTERNAL
> (find-symbol "SYMB4" "P3") ✠
P3::SYMB4
:INTERNAL
```

```
> (import 'p3::symb4 "P1") ✠
Error: Importing these symbols into the P1 package causes a name conflict:
(P3::SYMB4)
[condition type: PACKAGE-ERROR]
> (shadowing-import 'p3::symb4 "P1") ✠
T
> (package-shadowing-symbols "P1") ✠
(SYMB4 SYMB6)
```

If necessary, the `p2:symb4`, shadowed by the internal `symb4` in `"P1"` remains accessible via the colon notation. The next statements give the same value to both symbols, the '`=`' statement proves the *values* are the same, the `eq` statement proves both *symbols* are different; on the contrary, because `p1::symb4` was imported from `"P3"`, the last `eq`-test answers positively.

```
> (setf symb4 (setf p2:symb4 111)) ✠
111
> (= symb4 p2:symb4) ✠
T
> (eq 'symb4 'p2:symb4) ✠
NIL
> (eq 'symb4 'p3::symb4) ✠
T
```

It is even possible to add an *inherited* symbol to the shadowing list, maybe to declare a precedence with respect to another future rival symbol in another package intended to be used later. This assumes three different packages, one of them not yet used. The obvious experiment needs first to "detach" for example the package `"P3"` from `"P1"`, leaving on the contrary `"P2"` used by `"P1"`; remember `"P1"` is the current default package.

```
> (unuse-package "P3" "P1")
T
```

We then install two fresh external symbols with the same name in `"P2"` and `"P3"`:

```
> (export (intern "SYMB7" "P2") "P2") ✠
T
> (export (intern "SYMB7" "P3") "P3") ✠
T
```

As already seen with essentially the same example, if `"P1"` now decides to use `"P3"`, both symbols of name `"SYMB7"` collide:

```
> (use-package "P3" "P1") ✠
Error: Using package 'P3' results in name conflicts for these symbols: SYMB7
[condition type: PACKAGE-ERROR]
```

But because the symbol *name* `"SYMB7"` is unique in the collision, in some context, it could happen `"P1"` has no reason to directly use `p3:symb7` and would like to give in this situation a precedence to `p2:symb7`; this is done as follows:

```
> (find-symbol "SYMB7" "P1") ✠
SYMB7
:INHERITTED
> (shadowing-import 'symb7 "P1") ✠
T
> (use-package "P3" "P1") ✠
T
```

The `shadowing-import` implicitly cites `p2:symb7` which is external in a package used by `"P1"` and therefore inherited by `"P1"`; we see the use of `shadowing-import` solves the name conflict which otherwise would be raised by `use-package`. Let us examine our symbols as seen now from our three packages:

```
> (find-symbol "SYMB7" "P1") ✠
SYMB7
:INTERNAL
> (find-symbol "SYMB7" "P2") ✠
SYMB7
:EXTERNAL
> (eq 'symb7 'p2:symb7) ✠
T
> (find-symbol "SYMB7" "P3") ✠
SYMB7
:EXTERNAL
> (package-use-list "P1") ✠
(#<The P3 package> #<The P2 package> #<The COMMON-LISP package>)
```

The status have not been changed in `"P2` and `"P3"`, but seen from `"P1"`, the symbol is now *internal*, to obey the rules that any shadowing symbol in a package must be internal or external in this package, also a consequence of the implied importation.

16 A mathematical definition of the packaging system.

We recall the three general sets we must study:

- The set St of *strings*;
- The set P of *packages*;
- The set Sm of *symbols*.

with this difference: the set St is the *infinite* set of all the possible strings, while P and Sm are *finite* sets made of the packages and symbols previously *allocated* in the Lisp session we are working in.

We use the notations pn to denote the function `package-name` $pn : P \to St$ and sn to denote the function `symbol-name` $sn : Sm \to St$.

The map pn is *injective* for the packages; the function $pn : P \to pn(P)$ defines a 1-1 correspondence between the packages and their respective names. We forget here the nicknames of a package, a non-essential subject.

Every package P contains four *fundamental* lists:

- The list $InS(P)$ of its internal symbols;
- The list $ExS(P)$ of its external symbols;
- The list $ShgS(P)$ of its shadowing symbols;
- The list $U(P)$ of its used packages;

More precisely, these lists are lists of *pointers* to the mentioned objects, or if you prefer, of the machine addresses of these objects. It is important to understand these lists are made of (pointers to) *symbols*, not of their names. *Each* of these lists is *without any duplicates.*

The fundamental lists defined in a package are the source of a relatively rich terminology, detailed now. The set of symbols *present* in a package p, denoted by $PrS(p)$, is defined as the union $PrS(p) := InS(p) \cup ExS(p)$: a symbol is present in a package if it is internal or external in this package. This union is *disjoint*, the intersection $InS(p) \cap ExS(p)$ is required empty. Also the relation $ShgS(P) \subset Pr(P)$ is required: a shadowing symbol of the package p must be internal or external in this package: a shadowing symbol is present.

Let q be a package used by the package p. The symbols of q *shadowed* by a symbol of p are:

$$ShdS(q,p) = \{sm \in ExS(q) \ \underline{\mathbf{st}}\ \exists sm' \in ShgS(p) \ \underline{\mathbf{st}}\ sn(sm) = sn(sm')\}$$

In other words, a *precedence* relation is defined by the list $ShgS(p)$: if ever a symbol $sm' \in ShgS(p)$ and a symbol $sm \in ExS(q)$ have the same name, then, with respect to the package p, sm' takes precedence over sm, the symbol sm so is *shadowed.* If sm' is present in p but not shadowing, with $sm' \neq sm$, then a name conflict would be raised, not allowed.

The inherited symbols of a package p, denoted by $InhS(p)$, are defined as:

$$InhS(p) := \left\{\cup_{q \in U(P)}(ExS(q) - ShdS(q,p))\right\} - PrS(P)$$

A symbol is inherited in a package p if it is external in a *used* package q, not shadowed by a shadowing symbol of p, and not present in p. The union in the precedent formula is not necessarily disjoint: a symbol maybe external in several concurrent packages without causing a name conflict: only the *strings* cause name conflicts, not the symbols. A symbol could be simultaneously present in p and external in a package q used by p, requiring above the last term $-PrS(P)$ to define unambiguously what an inherited symbol is.

Finally the *accessible* symbols $AccS(p) := PrS(p) \cup InhS(p)$ are the symbols which are present or inherited; the very definition of $InhS(p)$ implies this union is disjoint. In this way, the *status* of a symbol with respect to a package p via which it is accessible is unambiguously defined as a keyword `:internal`, `:external` of `:inherited`. For a package $q \neq p$, no relation at all between the status of some symbol sm with respect to p and q; exercise: the sixteen pairs of possible status can be realized!

It is important to be lucid about possible intersections between the lists defined above. Let p be some package and q another package used by p. If $sm \in ExS(q)$ is an external symbol of q, the symbol sm *may* also be present in p, in which case its status $\boxed{\text{in } p}$ is internal or external, but not inherited. This in particular does not generate a name conflict, because it is the same symbol which is visible in two different packages via the same name, a symbol has only *one* name! Nevertheless its status *with respect to* p is not inherited, a symbol status is so deterministically defined, even if present in two packages communicating, one using the other one. Of course the same symbol has status external $\boxed{\text{in } q}$.

Following the same logic, if $q, q' \in U(p)$, the intersection $ExS(q) \cap ExS(q')$ is not necessarily empty.

The Lisp system maintains the *coherence* of these lists, defined as follows.

Definition. A set of packages is *coherent* if, for every package p, the function `symbol-name` defines a 1-1 correspondence $sn : AccS(p) \to sn(AccS(p))$.

This map is by definition surjective; it is injective if two different symbols accessible via p have different names. In this way the function `find-symbol` may work without any ambiguity; it searches first in the present symbols, and finally in the external symbols of the used packages; it is not necessary to compute the list of shadowed symbols: anyway, a shadow*ing* symbol, certainly in the present symbols, is found first. The coherence rule is mainly made to prevent the user from designing too contorted situations in name scopes, frequently sources of difficult bugs.

Several symbols with the same name may coexist in an environment. The most usual way to access a symbol consists in using its *name*; but *different* symbols may have the *same* name. In fact its name is not sufficient to access a symbol, a package must also be given; this can be the default package, in which case the direct notation without any colon is possible, or the package name can be explicitly used to prefix the symbol name, using the one or two colons notation. Of course the symbol so accessed must be *deterministically* defined. The definition just given says that a *unique* symbol of some name is accessible from some package.

The main function in the packaging organization is a map:

$$\rho : P \to [St \to ((Sm \cup \{\texttt{nil}\}) \times Stt)]$$

where Stt is a constant set of four symbols, the Lisp symbol `nil`, and three keywords, the possible status of an accessible symbol:

$$Stt = \{\texttt{nil}, \texttt{:internal}, \texttt{:external}, \texttt{:inherited}\}$$

Because of the assumed coherence of a package configuration, the function ρ, an avatar of the function `find-symbol`, given a package p and a string st, determines a *unique* symbol having the name *st* accessible via p, and also its status; it is possible also the search fails, in which case the function returns the pair (`nil`, `nil`)[6], the second `nil` being then a kind of null status: "nothing is accessible via p for the name *st*". The relation $\rho(p)(s) =$ (`find-symbol` p *st*) is always satisfied.

It is now simple to understand the various Lisp functions handling packages and symbols, in particular what name conflicts can be generated.

16.1 (intern *st* *p*)

This function first evaluates `(find-symbol` *st* *p*`)`. If a symbol is found, *both* values of `(find-symbol ...)` are returned.

If a symbol is not found, then Lisp *allocates* (creates in the memory space) a new symbol without any value, any function value. The `symbol-package` is initialized to p and this symbol is added to $InS(p)$. In this case `(values` *sm* `nil)` is returned, the second value `nil` being a code saying a new symbol has just been allocated.

No name conflict can be raised.

This function is implicitly used by the Lisp reader when it observes a symbol is present in the source text. If the colon syntax is not used, the default package is used for p, otherwise the (capitalized) package name of p is read before the colon(s), the (capitalized) symbol-name *st* after the colon(s), and the `(intern` *st* *p*`)` is executed, giving finally the symbol which is so *read* by the Lisp reader. In this way, the symbol can be a pre-existing one, or a just allocated consequence of this reading.

16.2 (unintern *sm* *p*)

A symbol *sm* is *interned* in a package p if it is present in this package, that is, if $sm \in PrS(p) = InS(p) \cup ExS(p)$, that is, if it is *present*. Uninterning a symbol asks for this membership relation not being anymore satisfied.

If $sm \notin PrS(p)$, nothing is changed and `nil` is returned. Otherwise *sm* is removed from $InS(p)$ or $ExS(p)$, and also from $Shg(p)$ if *sm* was shadowing with respect to p; in this case, `unintern` returns `t`. In the last case, when uninterning a shadowing symbol, a name conflict can be generated: it is possible $sm' \in ExS(q)$ and $sm'' \in ExS(q')$ have the same name and are *different* symbols in the respective packages q and q' used by p, and that this collision was resolved by the symbol *sm* shadowing both. If this happens, the `unintern` is rejected.

Also, if a symbol *sm* is uninterned from a package p and p is the home package of *sm*, then the home package of *sm* is modified and becomes `nil`, that is, the symbol becomes homeless; it is the only case where the symbol itself is modified by `unintern`. The same symbol could be external in another package used by p, in

[6]More precisely `(values nil nil)`.

which case the symbol *remains* accessible via p. Otherwise, if the home package was nil or another package, it is not modified.

16.3 (import sm p)

It is asked the symbol sm is to be added to $InS(p)$. Lisp first looks for a possible name conflict. This happens only if a *different* symbol $sm' \neq sm$ with the same name ($sn(sm) = sn(sm')$) is already accessible via p. If so, the importation is rejected.

If no name conflict is detected, Lisp acts for the importation. If the symbol is already *present*, it is let *unchanged* in $InS(p)$ or $ExS(p)$. Else, if the symbol is *inherited* because a package q used by p, then $ExS(q)$ is unchanged and sm is added to $InS(p)$ (not $ExS(p)$!). If sm is not accessible via p, it is simply added to $InS(p)$.

In the particular case of an imported homeless symbol, the symbol-package of the symbol is updated to p. It is the only case where a symbol object is itself modified by an importation. If the imported symbol is already owned by a package, whatever is its home package, it is not modified.

If the importation succeeds, the symbol will always be *present* in the target package, even if it was previously accessible only via a used package.

16.4 (export sm p)

The terminology “export” is not really appropriate, “externalize” would be more precise. The user of this statement asks Lisp to record the symbol sm in $ExS(p)$; allowed only if sm is accessible from p, in other words, the `export` function cannot *add* an accessible symbol to p. Taking account of the rule $InS(p) \cap ExS(p) = \emptyset$, if present in $InS(p)$, the symbol sm is also erased of the last list. If sm is external in a package q used by p, its status in q is unchanged, but sm is nevertheless added to $ExS(p)$.

Remember the `use-package` relation is not transitive, the fact $sm \in ExS(q)$ makes sm accessible via p using q but not via p' using p; if it is wished sm be accessible via p', it is possible to make it external in p by an export statement. Another solution of different nature would consist in importing sm in p', an operation not involving p.

After export, the symbol sm becomes accessible from any package q us*ing* p, that is, if $p \in U(q)$. This can generate a name conflict with a possible *different* symbol sm' of the same name present in q or accessible via q in another package $p' \in U(p)$, that is, if $sm' \in ExS(p')$. If this happens, the *export* statement is rejected.

16.5 (unexport *sm p*)

The symbol *sm* must be accesible via the package *p*. If the status of *sm* is *external*, it is downgraded to *internal*; otherwise it is unchanged, even if it is *inherited.*

16.6 (shadow {*sm* or *st*} *p*)

The first argument may be a symbol or a string, and if it is a symbol, only its symbol-name is used. So that we continue assuming the statement evaluated is `(shadow` *st p*`)`. First `shadow` executes `(import (intern` *st p*`))`; the `intern` component checks for a symbol accessible via *p* and interns one if a negative answer is obtained; the `import` component then adds the symbol to $InS(p)$ if the status was only *inherited.*

Now a symbol of name *st* is certainly internal or external in *p*; if not yet member of the shadowing list $Shg(p)$, it is added.

Exercise: this process cannot generate a name conflict.

16.7 (shadowing-import *sm p*)

It is intended to *import* the symbol *sm* in *p*, to record it in the shadowing list $Shg(p)$ of the package *p* and also to *resolve* any name conflict possibly raised by the importation.

This time the first argument must be a *symbol*, a pre-existing one or a symbol just allocated because of its presence in the source statement. The symbol is imported, see the function `import`, and added to the shadowing list $Shg(p)$ of *p* if not yet present in this list. This addition to the shadowing list solves the possible name conflicts with symbols inherited of packages used by *p*. But it is possible also a *different* symbol *sm*′ with the same name is beforehand *present* in *p*; if so, such a symbol is previously *uninterned* from *p*, that is, erased from the lists $InS(p)$, $ExS(p)$ and $Shg(p)$.

The difference between `shadow` and `shadowing-import` is clarified by the following toy example. We start a new Lisp session, allocate a package `"P"` and play with symbols of name `"A"` and `"B"` in packages `"USER"` and `"P"`.

```
> (defpackage "P") ✠
#<The P package>
> (setf list '(a b p::a p::b)) ✠
(A B P::A P::B)
> (shadow 'p::a "USER") ✠
T
> (shadowing-import 'p::b "USER") ✠
T
> (package-shadowing-symbols "USER") ✠
(B A)
```

```
> (eq 'a 'p::a) ✠
NIL
> (eq 'b 'p::b) ✠
T
> list ✠
(A #:B P::A B)
```

If we `shadow` the symbol `p::a` in `"USER"`, you observe the symbol `a` remains different from the symbol `p::a`. On the contrary, if we `shadowing-import` the symbol `p::b` in `"USER"`, then the symbols `b` and `p::b` are equal. In a sense, the statement `(shadow 'p::b "USER")` is cheating because of the presence of '`p::`' which has no role at all, for only the *name* of the symbol `p::a` is used, and this name is the same as the name `"A"` of `a`, that is, because of the default package, `user::a`. So that finally the symbol defined by `"A"` via the package `"USER"` is unchanged, in particular remains different from the symbol `p::a`.

On the contrary, if `shadowing-import` is used, an `import` operation is induced, so that the symbol `p::b` becomes directly accessible via `"USER"`; the previous symbol $\rho($`"USER"`$)($`"B"`$)$ has been therefore *uninterned* and is now unreachable by `find-symbol`. But it is still visible in the list located by `list`, as the homeless symbol `#:B`.

The end result of our test is :

- The symbol `user::a` is unchanged but added to $ShgS($`"USER"`$)$.
- The symbol `user::b` is almost unchanged, but no more accessible via `"USER"`; the only change is its symbol-package become `nil`, because the symbol has been uninterned from its home-package.
- No change at all for `p::a`, only its name has been used for something which does not concern it.
- No change for the symbol `p::b` except that it is now directly accessible via `"USER"` and cited now as `B` in the value of `list`.

16.8 (use-package q p)

Unless q is already in $U(p)$, the package q is added to the `package-use-list` of p.

Lisp checks before for possible name conflicts. A symbol $sm \in PrS(p)$ could collide with a *different* symbol $sm' \in ExS(q)$ if they have the same name $sn(sm) = sn(sm')$. On the contrary, if $sm = sm'$, no name conflict is generated. You cannot resolve such a name conflict by adding sm' to $ShgS(q)$, hoping to give the precedence to the new inherited symbol: shadowing operates in a unique direction, from a using package toward a used package; if you do so by a `shadowing-import` of sm' into p, the symbol sm would be coherently uninterned, erased from $PrS(p)$, and in a sense replaced by sm'. On the contrary, you can prepare a precedence of sm over sm' before the `use-package` by previously *shadow* sm in p. Also Lisp can propose itself to *unintern* the symbol sm in a *continuable* error to resolve such an observed name conflict.

Another type of name conflict could be raised if another package q' is already used by p; it is then possible two *different* symbols $sm \neq sm'$ with the same name st are respectively external in q and q', in which case $\rho(p)(st)$ is ambiguous; if so, the `use-package` statement generates an error. On the contrary, the intersection $ExS(q) \cap ExS(q')$ may be non empty: a symbol external in both packages does not generate any ambiguity via p. As usual, always important to remember the lists such as $ExS(q)$ and $ExS(q')$ are lists of *symbols*, not strings. A name conflict can be generated only by *different* symbols having the same `symbol-name`.

If no name conflict is detected, the package q is added to $U(p)$, so that the new set of accessible symbols via p is:

$$\text{new-}AccS(p) = \text{old-}AccS(p) \cup (ExS(q) - ShdS(q,p))$$

16.9 (unuse-package q p)

If the package q is in $U(p)$, it is removed from this list.

16.10 A bug in package-internal-symbols?

When preparing these notes, the author thought a moment having met a mysterious bug. The minimal model for this bug goes as below in a new Lisp session

```
> (defun package-internal-symbols (package)
    (let ((rslt nil))
      (do-symbols (s package)
       (when (eq (second
                   (multiple-value-list
                     (find-symbol (symbol-name s) package)))
                 :internal)
         (push s rslt)))
      rslt)) ✠
PACKAGE-INTERNAL-SYMBOLS
> (defpackage "P1") ✠
#<The P1 package>
> (defpackage "P2") ✠
#<The P2 package>
> (use-package "P2" "P1") ✠
T
> (export 'p2::a "P2") ✠
T
> (import 'p2:a "P1") ✠
T
> (package-internal-symbols "P1") ✠
(P2:A P2:A)
```

Our function `package-internal-symbols` is redefined, the packages `"P1"` and `"P2"` are allocated, `"P1"` using `"P2"`. A symbol of name `"A"` is made external in `"P2"` and then imported in `"P1"`. When we ask for $InS($`"P1"`$)$, we get two symbols `P2:A`, why?

The point is that the standard Lisp function `do-symbols` runs through all the symbols *accessible* via `"P1"`, but does not check for possible *duplicates*. Ah! it is possible a symbol is duplicate in $AccS($`"P1"`$)$!? Does not seem compatible with the exclusion of duplicates in general in $InS(p)$ and $ExS(p)$ and also the rule $InS(p) \cap ExS(p) = \emptyset$. Yes but a – one !– symbol can be simultaneously in $InS(p)$ and $ExS(q)$ for a package q used by p, the case here for our symbol `p1::a = p2:a`.

And (`do-symbols` (s p) ...) makes `s` run through all the internal symbols of p, then through all the external symbols of p, and finally through all the external symbols of q for every package q used by p. It happens our symbol is in $InS($`"P1"`$)$ because imported there, and also in $ExS($`"P2"`$)$ from where it was initially defined. Yet the programmer of `package-internal-symbols` had ordered to check the status of the symbol, so that the (fake) second symbol should have been rejected? No, it is interesting to see the path followed by Lisp: only the symbol-name is used, and the examined package, so that when Lisp computes `(find-symbol (symbol-name s) package)`, it obtains *the* symbol, in fact the only one in this story, the symbol and its status with respect to the `package` argument, that is in our case the package `"P1"` and the answer is `:internal`, even if at this time `s` $\in ExS($`"P2"`$)$.

This illustrates the traps that can be opened by different packages seeing the same symbol in various ways. Remember: no name conflict can be caused by one symbol, a name conflict happens only if *two* different symbols with the same name are "visible" by the same package. Nothing of this sort here, which does not prevent *one* symbol from being visible by several packages.

To overcome this problem, using the Lisp function `delete-duplicates` is enough.

```
> (defun package-internal-symbols (package)
    ... ...
        (push s rslt)))
      (delete-duplicates rslt))) ✠
PACKAGE-INTERNAL-SYMBOLS
> (package-internal-symbols "P1") ✠
(P2:A)
```

16.11 Symbol-package revisited.

We insisted about the fact that this function is not really interesting. To be complete, we *prove* that a symbol $sm \in PrS($`(symbol-package` sm`)`$)$; in other words, if the symbol-package of the symbol sm is p, then sm is present in p, either internal or external. When a symbol is allocated, necessarily in fact via `intern`, it is in particular... interned in the package being the second argument of `intern`, so that it becomes a member of $InS(p)$. It may be exported in this package, becoming a member of $ExS(p)$, but remains present. Via `import` or `shadowing-import` the symbol can be added in $InS(q)$ for other packages q and also then upgraded to $ExS(q)$ in the same packages, but the membership in the initial package $PrS(p)$

is not modified.

The only way to change this situation is under `unintern` when the package quoted by `unintern` again is *p*; then the symbol becomes homeless. Situation which can be modified later again with `import` and `shadowing-import` in which case the symbol is then interned in this possibly new package, and the story continues in the same way. Finally a symbol always is present in its home-package, unless it is homeless.

So that, as it was mentioned in the introduction, the `symbol-package` qualifier of a symbol does divide the set of *interned* symbols into packages. It is not possible to have two different interned symbols with the same symbol-name and also the same symbol-package, for they would be interned in the same package, with the same name, and the set of packages would not be *coherent*.

An interned symbol is internal or external in its symbol-package, certainly visible via this package. But it may be visible from many other packages, and this information cannot be read inside the symbol. If it is external, it is accessible via the packages which use its package; the list of these packages can be determined by the standard Lisp function `package-used-by-list`. But a symbol can also be imported by other packages, which packages may decide to make external (the pointer to) this symbol and so visible from other packages... Exercise: try to write down a function `symbol-accessible-from` needing one argument, a symbol, and returning the list of all packages where this symbol is accessible; do not forget the contorted particular case of `lisp:nil`! You cannot avoid the use of the standard Lisp function `list-all-packages`.

An arbitrary number of homeless symbols may have the same name, as illustrated by :

```
> (eq (print (make-symbol "Z")) (print (make-symbol "Z"))) ✠
#:Z
#:Z
NIL
```

When `make-symbol` is called, Lisp allocates a *new* homeless symbol with the prescribed name, even if another one with the same name has been already allocated. You see *both* homeless symbols allocated above are output in the same way, but yet are *different.*

17 Practical uses of the packaging system.

Most often the esoteric technichalities detailed before have no reasonable uses in concrete programming. But they can help to understand mysterious behaviours of your Lisp system.

A particularly irritating fact showed by some systems is the following. When you install, not load, only install a source file in the editor of your Lisp IDE, these systems immediately explore the symbols included in these files and intern them in a package, sometimes the package indicated in an inital `(in-package "XXX")` of the file, or still more inappropriately, in the standard `"USER"` package, or also in the current default package, even if the initial `in-package` cites another package. This leads later in symbol collisions sometimes hard to identify. A developer who meets mysterious symbol collisions is advised to search first in this direction. In this case, you should make independent micro-experiments to study the actual action of your IDE when you install a file in your development environment. Probably in the old times of Lisp, this was done to improve the efficiency, but in the modern configurations, it is impossible to see sensible reasons justifying such a strategy.

Most symbols of a Lisp file in fact are symbols with lexical scope, a scope limited to a small part of the code, typically *inside* a function, a `let` statement, a `do` statement and so on. These references are converted by the compiler into rough machine addresses, and in fact these symbols have no real life at running time.

The most important symbols who have a real life at runtime are those who locate functions, those defined through a `defun` statement and the similar ones. If a large program is divided in several files, then some decisions must be taken about the status of such a symbol.

Usually, a correspondance "file $\leftrightarrow$ package" is defined. Several Lisp systems require a file begins by the appropriate `(in-package ...)` statement, explaining all the symbols used in this file are to be understood as present in this package, except when something else is a consequence of decisions taken outside this file or even in this file.

In the Ansi definition of `in-package`, the quoted package must be already allocated. So, beginning a file by:

```
(defpackage "MY-PACKAGE")
(in-package "MY-PACKAGE")
```

is tempting but does not obey the requirement about the first instruction of a file being the appropriate `(in-package ...)`.

The solution is to have a first file in the project which could be called `packages.cl` and containing all the necessary `(defpackage ...)` of the project. The `defpackage` statement is flexible and can contain also all the main informations about this package, in particular defining the list of its external symbols and its used packages, see the Ansi definition of `defpackage`. This file is so a good base to structure the "public" documentation, explaining the role of these packages and external symbols.

Along the same line, a possibility can consist also in having a package `"EXTERNAL"` containing all the external symbols of the programmed application. In this way, the other packages may systematically use this package, giving to this package a status similar to the status of the `"LISP"` package. Which does not prevent another package for specific reasons to import some symbols of other packages, symbols whose role is too specific to justify to make them "globally" external.

18 Shortcuts.

Several possibilities are given to the developer to shorten the Lisp statements handling the packaging system.

To be strict in our explanations and close to the deep structure of Lisp, we always gave for a package a string as its identifier. A symbol can be used instead, its name then being used as the intended string. For example `(defpackage :my-package)` is equivalent to `(defpackage "MY-PACKAGE")`, for the name of the keyword `:my-package` is used, and because of the capitalization rule of the symbols applied by the Lisp reader, the name of `:my-package` is `"MY-PACKAGE"`, which through the `defpackage`, becomes a package-name following the standard rule asking for an uppercase name.

Also, each time a package argument is necessary, most often the second one, it can be omitted if this package is the current default package, justifying this terminology. When using these shortcuts, do not forget the matter of readability of the source text.

-o-o-o-o-o-o-